MASTER EN SCIENCES INFORMATIQUES

Comment faire adopter à un personnage virtuel les mouvements d'un être humain ?

Delannay, Gaetan

Award date:
1998

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 1997-1998

Comment faire adopter à un personnage virtuel les mouvements d'un être humain ?

Gaëtan Delannay

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique.

Ce travail aborde le problème de la capture de mouvements, qui consiste à faire adopter à un personnage virtuel les mêmes mouvements que ceux effectués par un être humain.

Ce problème est tout d'abord envisagé dans le cadre du projet GabbyCapture qui s'est déroulé dans l'entreprise Neurones Animation SA, active dans les domaines du dessin animé et de l'image de synthèse. La réalisation principale du projet a été l'écriture d'une extension au logiciel Gabby, interne à Neurones, pour lui permettre de traiter les données générées par un système de capteurs MotionStar de la firme Ascension Technologies. Gabby est la partie logicielle d'un système qui permet d'animer des personnages virtuels en temps réel ; le système MotionStar est un système magnétique qui mesure la position et l'orientation de capteurs que l'on peut fixer sur un animateur humain.

Une discussion est ensuite entamée au sujet des méthodes de développement de systèmes temps-réel. L'approche qui a été adoptée a consisté à présenter une spécification puis une architecture, réalisée avec le langage ROOM, de l'extension à Gabby dont le code avait été écrit lors de la réalisation du projet. L'objectif de cette démarche est de montrer les avantages qu'une méthode de développement top-down *et l'utilisation d'un outil d'analyse spécifique peuvent apporter pour l'écriture de programmes temps-réel.*

This work deals with the problem of motion capture, which consists in applying on a virtual character the movements made by a human being.

This problem is first tackled within the GabbyCapture project, realized in the firm Neurones Animation SA, active in the fields of cartoons and computer graphics. The main activity in this project was to develop an extension to the software Gabby, internal to Neurones, in order to allow it to process the data generated by a MotionStar motion capture system, from the firm Ascension Technologies. Gabby is the software part of a system which allows to animate virtual characters in real time. The MotionCapture system is a magnetic system which measures the position and orientation of sensors that one can fix on a human body.

After this, a discussion is started on concerning the engineering methodologies for developing real-time systems. The approach was to present a specification and a software architecture, realized with the ROOM language, of the Gabby's extension of which the code was written during the GabbyCapture project. The objective is to show the advantages of the adoption of a top-down methodology and the use of a specific design tool for writing real-time applications.

Remerciements

Je tiens à remercier le promoteur de ce mémoire, Claude Cherton, pour ses précieux conseils et pour l'intérêt constant qu'il a porté à la réalisation de ce travail.

Je tiens également à remercier mes amis et ma famille pour leur aide et leur patience.

Introduction

« Comment faire adopter à un personnage virtuel les mouvements d'un être humain ? »

L'objectif de ce travail est de répondre à cette question. Nous allons voir que nous lui donnons deux sens différents : ce n'est donc pas à une, mais à deux questions que nous allons tenter de répondre. Avant de le faire, il est nécessaire de les distinguer et de déterminer la portée exacte de chacune d'elles.

Du 1er septembre 1997 au 31 janvier 1998, j'ai effectué un stage dans l'entreprise liégeoise Neurones Animation SA.

Cette société est active dans deux domaines : celui du dessin animé et celui de l'image de synthèse[1]. Dans ce dernier, Neurones réalise, entre autres, des «cyberacteurs» : ce sont des «marionnettes virtuelles» qui peuvent être animées en temps réel. L'animateur humain «tire les ficelles» en parlant dans un micro —ce qui a pour effet de faire bouger les lèvres du cyberacteur— et en déplaçant un stylet sur une tablette graphique, pour faire adopter à la marionnette des expressions faciales, et, dans une moindre mesure, des mouvements corporels.

Vers le milieu de l'année '97, Neurones a décidé de pouvoir animer ses personnages (cyberacteurs et personnages de dessin animé) sur base de mouvements effectués par des êtres humains. Le stage que j'ai effectué dans cette entreprise concernait ce sujet.

Ces informations nous permettent d'introduire un premier sens à la question initiale. Celle-ci peut, prise dans ce sens, se reformuler de la manière suivante : « Comment appliquer les mouvements d'un être humain sur les personnages virtuels créés par Neurones Animation SA ?»

[1] Par «image de synthèse» on entend les techniques d'imagerie parfois appelées «techniques 3D», que l'on retrouve fréquemment dans les effets spéciaux au cinéma, dans la publicité ou dans les jeux vidéo. Nous donnerons ultérieurement une définition précise de l'image de synthèse.

La réponse à cette question s'est concrétisée dans le projet 'GabbyCapture', qui s'est déroulé chez Neurones du 1er septembre 1997 au 31 janvier 1998. Pour pouvoir le réaliser, la société liégeoise a acheté un système de capteurs, dispositif qui permet d'enregistrer des informations de base concernant les mouvements d'un être humain. Plus précisément, il est constitué d'un certain nombre de capteurs, qui sont capables de mesurer leur position et leur orientation, et que l'on fixe à proximité des articulations d'un être humain. Les données générées par les capteurs peuvent être mises à disposition d'un ordinateur.

Puisque c'est le logiciel Gabby, développé par Neurones, qui permet d'animer en temps réel les cyberacteurs, et qu'il pourra aussi, comme nous le verrons, jouer un rôle dans l'animation des personnages de dessin animé, une partie du projet GabbyCapture a donc consisté à écrire une extension à Gabby, pour que celui-ci puisse, d'une part, récupérer les données provenant du système de capteurs et, d'autre part, animer un cyberacteur de telle sorte qu'il réalise, en temps réel, les mêmes mouvements que ceux de l'être humain sur lequel les capteurs sont fixés.

Des logiciels commerciaux sont aussi utilisés chez Neurones pour animer les personnages virtuels : une deuxième partie du projet a donc consisté à rechercher et étudier les «pilotes» existant sur le marché, pouvant réaliser l'interfaçage de ces logiciels avec le système de capteurs de Neurones.

Deux personnes, dont moi-même, ont été chargées de réaliser le projet GabbyCapture.

Nous pouvons maintenant affiner le premier sens que nous donnons à la question initiale, et reformuler celle-ci, dans ce sens, de la manière suivante : « Quels ont été les objectifs précis du projet GabbyCapture et quelles ont été les réalisations auxquelles il a donné lieu ? »

Une partie de la réalisation du projet a consisté à développer l'extension à Gabby, appelée, comme le projet, GabbyCapture [1]. Celui-ci a rempli partiellement l'objectif qui lui était assigné et permet d'animer, dans une certaine mesure, un cyberacteur sur base des mouvements enregistrés par le système de capteurs. Pour créer GabbyCapture, la méthode de développement qui a été adoptée, directe et rapide, consistait principalement à écrire du code en réalisant un minimum d'étapes préalables. Dans la suite de ce travail, ce développement sera resitué dans son contexte et les raisons de l'adoption de cette méthode seront expliquées. Durant le développement de GabbyCapture, je me suis néanmoins demandé si une autre méthode n'aurait pas pu être adoptée et, si oui, quelles auraient été ses répercussions sur la réalisation du projet.

Ces interrogations ont mené au deuxième sens que nous donnons à la question initiale, et qui a trait à la méthode de développement de GabbyCapture. La question initiale, dans ce sens, peut être reformulée comme suit : «Quelle méthode de développement adopter pour l'écriture d'un logiciel, qui, à partir des données provenant d'un système de capteurs, anime en temps réel un personnage virtuel?»

Après la période de stage, j'ai réalisé une spécification du problème auquel GabbyCapture, tel que nous l'avons développé, apporte une solution. J'ai ensuite écrit l'architecture du logiciel, avec le langage ROOM dont nous reparlerons, en me basant sur le code de GabbyCapture, mais en ayant déjà écrit sa spécification. Ce processus de réécriture de GabbyCapture m'a permis de comparer deux méthodes de développement : une méthode *top-down* consistant à réaliser une spécification et une architecture à l'aide d'un outil d'analyse, avant d'écrire du code, et la méthode de développement que nous avions utilisée chez Neurones.

[1] «Capture» pour «capture de mouvements».

Nous pouvons maintenant affiner le second sens que nous donnons à la question initiale, et reformuler celle-ci, dans ce sens, de la manière suivante : « Quels avantages peuvent apporter une démarche de développement *top-down* et l'utilisation d'un outil d'analyse spécifique pour l'écriture d'un logiciel comme GabbyCapture ? »

Nous avons donc précisé les deux sens de la question initiale et, partant, délimité les réponses qui vont pouvoir être apportées dans ce travail.

Le problème de la «capture de mouvements» ne sera donc abordé que dans les limites de la réalisation du projet GabbyCapture. Plutôt que d'approfondir ce problème, nous avons préféré examiner de plus près la méthode de développement de GabbyCapture.

Concernant l'étude de cette méthode, nous avons adopté une approche «pratique», consistant à écrire une spécification et une architecture du logiciel qui avait été développé dans le cadre du stage. L'objectif était de pouvoir tirer des conclusions sur base de l'expérience de la réalisation de deux démarches de développement différentes pour un même problème.

Nous avons aussi posé un cadre plus «théorique» autour de ce travail, principalement en ce qui concerne les aspects graphiques de l'informatique.

Ce travail est divisé en 5 chapitres, comporte une annexe «papier» et une seconde annexe qui se trouve sur la disquette ci-jointe.

Le premier chapitre introduit les notions de base de l'informatique graphique, ou «infographie». Dans ce chapitre, nous partirons de l'explication d'un modèle de système graphique. Nous développerons ensuite un paradigme graphique largement répandu, qui consiste à s'inspirer du fonctionnement des systèmes optiques pour produire des images par ordinateur, et qui a façonné les systèmes graphiques tels qu'on les retrouve aujourd'hui. Nous terminerons en abordant les applications de l'infographie.

Le second chapitre introduit la société liégeoise Neurones Animation SA. Après avoir parlé du groupe Neurones dont elle fait partie, nous aborderons sa structure, ses activités et ses projets futurs.

Le troisième chapitre présente le projet GabbyCapture : ses objectifs, ainsi que les réalisations auxquelles il a donné lieu —l'écriture de GabbyCapture, ainsi que la recherche et l'étude des pilotes pour le système de capteurs.

Le quatrième chapitre contient la spécification de GabbyCapture, que nous avons réalisée après l'écriture du logiciel. Après avoir défini un modèle de personnage virtuel, nous modéliserons ce qu'on entend par «mouvements d'un être humain» et définirons ensuite la manière dont ces mouvements peuvent être appliqués à un personnage virtuel.

Le cinquième et dernier chapitre présente l'architecture de GabbyCapture, que nous avons réalisée après le code et aussi après la spécification du chapitre 4. Dans un premier temps, nous présenterons le langage avec lequel elle a été écrite, ROOM (*Real-time Object-Oriented Modeling*) ainsi que les raisons pour lesquelles nous avons choisi ce langage. Nous exposerons ensuite, de manière textuelle, notre architecture ROOM, puis nous l'évaluerons, en critiquant et en justifiant les choix qui ont été effectués lors de sa réalisation. Cette évaluation correspondra à une comparaison entre les deux démarches de développement qui ont été mises en présence l'une avec l'autre.

Nous conclurons en reprenant les réponses qui ont été apportées à notre question de départ, qui forme le titre de ce travail.

La première annexe présente la version complète, en langage ROOM [1], de l'architecture qui est introduite au chapitre 5.

La seconde annexe, qu'on peut trouver sur la disquette qui accompagne ce travail, contient le code de GabbyCapture. Un fichier Lisez-moi.txt, présent dans son répertoire racine, expose plus précisément son contenu.

[1] Nous verrons que certaines parties de l'architecture n'ont pas été modélisées entièrement en ROOM.

Table des matières

Annexe A1

Architecture ROOM de GabbyCapture **91**

Chapitre 1
L'infographie

L'infographie [1] est définie comme l'ensemble des outils, techniques et méthodes qui permettent de générer des images en utilisant des moyens informatiques.

Deux disciplines différentes cohabitaient initialement sous ce terme. La synthèse d'images, d'une part, concernait la production d'images à partir de modèles informatiques; l'analyse d'images (*image processing*), d'autre part, dénotait le processus inverse, par lequel des images réelles étaient transformées en modèles informatiques pour pouvoir être traitées de diverses manières.

Il fut un temps où l'identification de deux branches distinctes prenait son sens : techniques, méthodes et même outils étaient dissemblables. Par exemple, alors que l'analyse d'images utilisait la technologie *raster* pour les écrans, la synthèse d'images utilisait la technologie vectorielle.

Aujourd'hui les différences se sont amenuisées. La technologie *raster* s'est étendue aux deux domaines ; de plus, les programmes d'analyse utilisent des images de synthèse et les programmes de synthèse utilisent des images acquises. Dans ce contexte, nous définissons l'infographie comme l'étude des processus, qui, à partir de modèles informatiques —obtenus au départ d'images existantes ou créés de toutes pièces avec des outils de conception graphique—, génèrent des images informatiques.

Ce premier chapitre comprend trois sections :

- la première présente les outils infographiques au travers d'un modèle de système graphique. Ce modèle est assez général pour pouvoir schématiser la plupart des systèmes existants, quelle que soit leur taille ou leurs fonctionnalités spécifiques.

- La seconde pose les bases d'un paradigme graphique, le *modèle de la caméra synthétique*, qui fait quasi l'unanimité aujourd'hui. Les méthodes et les techniques issues de ce courant ont façonné les systèmes tels qu'ils sont implémentés à l'heure actuelle.

- La troisième reprend les principales applications de l'infographie et les organise en quatre catégories: affichage d'informations, conception, simulation et interfaces hommes-machines.

[1] L'«infographie» ou «informatique graphique» se traduit en anglais par *computer graphics*.

1.1. Un modèle de système graphique

La figure 1.1 présente les différents éléments *hardware* et *software* d'un système infographique ainsi que la manière dont ils s'enchaînent pour produire une image.

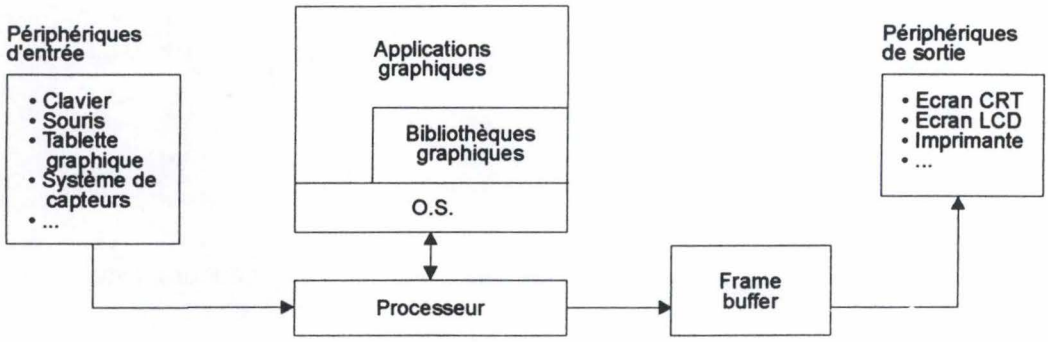


Figure 1.1 Un modèle de système graphique

Périphériques de sortie □ La plupart des écrans actuels, qu'ils soient à tube cathodique (CRT: *cathode-ray tube*) ou à crystaux liquides (LCD: *liquid-crystal display*), ainsi que la majorité des imprimantes, font partie de la famille des *raster outputs* : les images qu'ils sont capables d'afficher doivent leur être fournies sous la forme de «trames», ou tableaux de pixels [1]. Le modèle de la figure 1.1 est basé sur cette technologie *raster*. Il existe aussi des périphériques de sortie vectoriels, plus marginaux, comme les tables traçantes ou certains écrans, pour lesquels les images doivent être spécifiées sous forme de vecteurs.

Frame buffer □ Chaque pixel d'une image *raster* est codé sur un certain nombre de bits. Le *frame buffer* est la mémoire dans laquelle ces bits sont stockés avant d'être envoyés sur l'output. Il contient les informations nécessaires à l'affichage d'une image entière (une *frame*). La taille et la qualité d'une telle image sont fonction des caractéristiques du frame buffer :

- sa résolution —son nombre total de pixels (1024x768, par exemple),
- sa profondeur —le nombre de bits utilisés pour coder un pixel, déterminant le nombre de couleurs qu'il est possible d'assigner à ce pixel (24, par exemple).

Processeur □ L'image imprimée sur un écran disparaît après quelques millisecondes: il s'agit donc de «rafraîchir» celui-ci continuellement, et ce à un taux suffisant pour que l'oeil humain perçoive cet enchaînement comme une image fixe [2]. Les processeurs des systèmes graphiques rudimentaires étaient vite saturés par l'exécution en boucle de la routine d'affichage à l'écran. Le premier type de matériel informatique construit spécifiquement pour les besoins de l'informatique graphique a été le processeur graphique. Au départ, sa fonction était de décharger le processeur principal du travail de rafraîchissement. Son champ d'action s'est ensuite agrandi : son jeu d'instructions et ses circuits se sont développés pour pouvoir exécuter «en hardware» des primitives

[1] Le mot «pixel», né de la concaténation des mots «*picture*» et «*element*», désigne l'unité élémentaire constitutive d'une image informatique.

[2] Pour un écran dont la résolution est de 1024x768, le taux de rafraîchissement à partir duquel l'être humain ne perçoit plus de scintillement est de ± 70Hz.

graphiques plus complexes, comme l'affichage de polygones. La figure 1.2 montre une évolution possible du modèle de base par rapport à ces changements. Le processeur principal envoie les instructions graphiques (la *display file*) dans une mémoire à laquelle le processeur graphique peut accéder ; celui-ci les exécute pour former l'image dans le frame buffer.

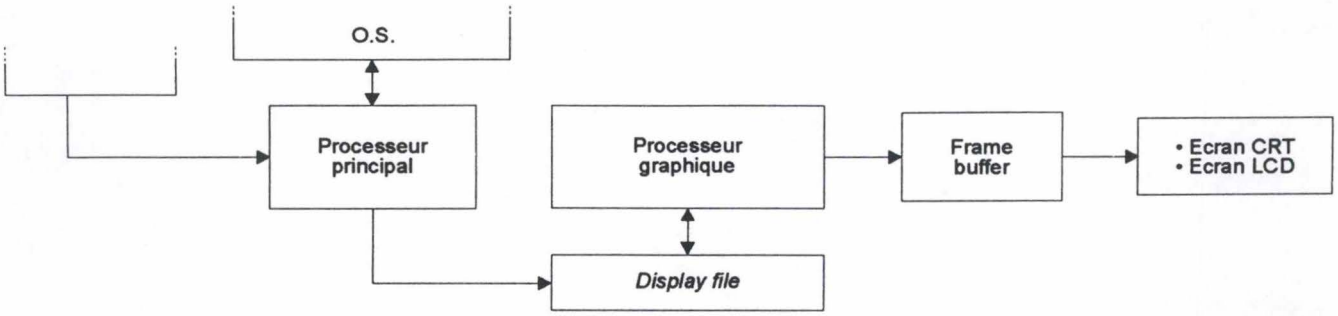


Figure 1.2 Le modèle de base étendu avec un processeur graphique

Bibliothèques graphiques □ La nécessité de disposer de fonctions standards permettant au programmeur de manipuler des concepts graphiques plus intuitifs et plus pratiques que les tableaux de pixels a entraîné l'apparition de bibliothèques graphiques. Les plus courantes aujourd'hui sont *PHIGS* (*Programmer's Hierarchical Interactive Graphics System*), *GKS* (*Graphical Kernel System*) et *OpenGL* (*Open Graphics Library*). On verra par la suite la nature exacte des concepts dont elles sont les implémentations.

Applications graphiques □ Comme nous le découvrirons dans la troisième section de ce chapitre, des applications graphiques ont été développées dans un grand nombre de domaines et forment aujourd'hui une part importante du parc des programmes informatiques. Les interfaces de la plupart des systèmes d'exploitation actuels en sont l'exemple le plus répandu. Ils sont par ailleurs à l'origine d'un trait majeur des systèmes graphiques : l'interactivité.

Périphériques d'entrée □ Pour pouvoir interagir avec leur environnement, les systèmes graphiques disposent d'un grand nombre de périphériques d'entrée : souris, clavier, tablette graphique, dataglove, joystick, systèmes de capteurs, etc.

Les premiers systèmes graphiques ont commencé avec l'affichage de données sous forme textuelle sur un écran à tube cathodique, et avec la sortie de caractères sur imprimante. Ils correspondent donc aux premiers ordinateurs. Les systèmes actuels sont capables de produire des images photoréalistes et des images animées en temps réel pouvant jusqu'à recréer virtuellement de manière assez convaincante un environnement visuel complexe.

1.2. Le modèle de la caméra synthétique

L'idée, pour produire des images par ordinateur, est de s'inspirer du fonctionnement des systèmes optiques, comme le système visuel humain ou la caméra. Dans ce sens, créer une image informatique est un processus qui est vu comme similaire à celui de créer une image via un système optique. Ce paradigme est connu sous le nom de «modèle de la caméra synthétique» et est à la base

de la plupart des systèmes graphiques actuels. PHIGS, GKS et OpenGL s'inspirent de ce modèle.

Cette section aborde dans un premier temps (1.2.1) les systèmes optiques. Puisque, comme nous le verrons ci-après, tout processus de formation d'images met en scène trois types d'acteurs —les sources de lumière (et la lumière qu'elles génèrent), les objets et les observateurs (les systèmes optiques)—, nous allons d'abord présenter de manière générale ces acteurs et les relations qu'ils entretiennent dans un processus de formation d'images (1.2.1.1) avant de présenter deux «observateurs» particuliers : le système visuel humain (1.2.1.2) et la caméra ponctuelle (1.2.1.3).

La seconde partie de cette section (1.2.2) aborde la manière dont, en infographie, on s'inspire des systèmes optiques pour créer des systèmes infographiques. Après avoir présenté un modèle informatique de la couleur (1.2.2.1), on s'attardera sur la modélisation infographique des sources lumineuses, des objets et des observateurs (1.2.2.2). Nous terminerons par la présentation du processus de formation d'images proprement dit (1.2.2.3).

1.2.1. Systèmes optiques

1.2.1.1. Sources de lumière, objets et observateurs

Des sources de lumière illuminent le «monde réel» de leurs rayons. Certains de ceux-ci, après avoir éventuellement frappé des «objets» de ce monde, sont susceptibles d'être (re)dirigés vers un observateur (un système optique) qui peut alors produire une image.

La lumière est une forme de radiation électromagnétique. Elle se se propage sous la forme d'ondes dont les longueurs d'ondes sont comprises entre 350 et 780 nm. A chaque longueur d'onde correspond une couleur. Un rayon lumineux peut transporter de la lumière monochromatique (constituée d'une seule longueur d'onde) ou de la lumière complexe (constituée de plusieurs longueurs d'ondes).

Quand un rayon lumineux vient frapper un objet,

- une partie de la lumière incidente est **réfléchie** : elle «rebondit» sur la surface de l'objet.
  - Si la surface de l'objet est parfaitement polie, le rayon réfléchi résultant est dévié selon un angle de réflexion qui est égal à l'angle d'incidence (figure 1.3 [a]) ;
  - si la surface n'est pas parfaitement polie, la surface peut **diffuser** le rayon et donner naissance à un ensemble de rayons réfléchis (figure 1.3 [b]).

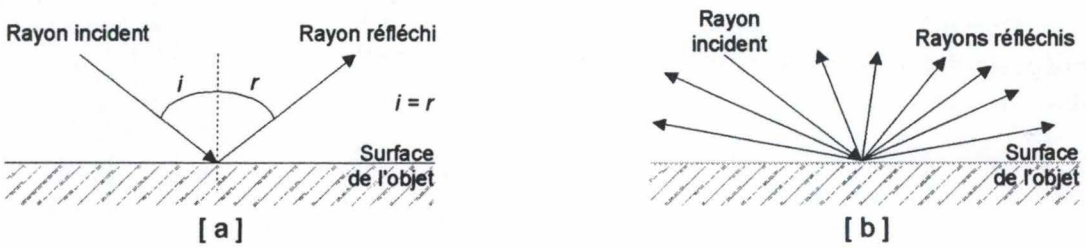


Figure 1.3 Lumière incidente réfléchie

- Une partie de la lumière incidente est **absorbée** par la surface de l'objet et est transformée en chaleur.

• une partie de la lumière incidente est **transmise** : elle passe au travers de l'objet. Pour autant que la surface de l'objet soit bien polie,

- si la lumière incidente est monochromatique, le rayon transmis résultant est dévié selon un certain angle de réfraction (figure 1.4 [a]) ;
- si la lumière incidente est complexe, la réfraction n'est pas la même pour toutes les radiations qui composent le rayon incident : il y a **dispersion**, comme cela est apparu dans l'expérience de Newton (figure 1.4 [b]). Cette expérience, détaillée dans [Fle68], consiste à faire passer de la lumière blanche par une petite ouverture circulaire O pour lui faire traverser un prisme P, dont les faces IJ et JK sont polies. La lumière aboutit sur l'écran E: on constate qu'au lieu de former une tache circulaire blanche, comme cela se produirait en l'absence de prisme, elle s'étale en une bande colorée VR allant du violet (le plus dévié) au rouge (le moins dévié) par l'intermédiaire du bleu, du vert, de l'orange et du jaune. Un trou O' laisse passer la lumière d'une seule couleur, verte, par exemple, qui traverse un second prisme P' et est reçue en T sur un second écran E'. On constate qu'elle est à nouveau déviée, mais que sa couleur n'est pas modifiée par la traversée du second prisme et qu'elle n'est pas étalée (du moins si O' est suffisemment petit).

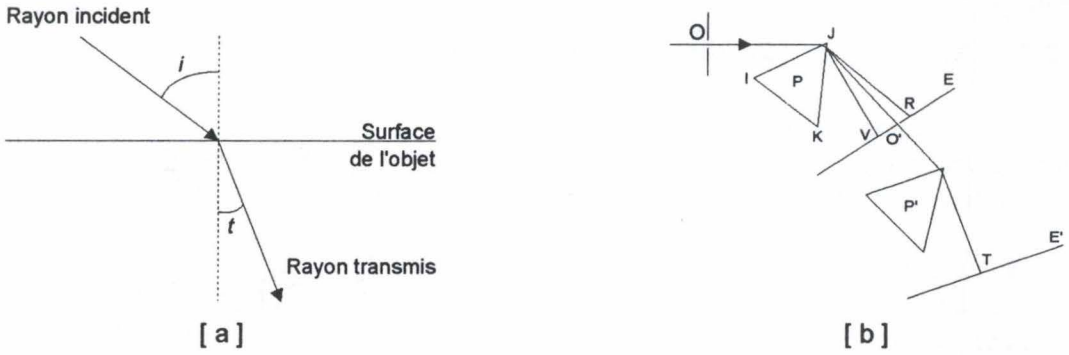


Figure 1.4 Lumière incidente transmise

La répartition d'un rayon incident en lumière réfléchie, diffusée, absorbée, transmise et dispersée dépend de la nature de ce rayon et de la surface de l'objet. Une surface parfaitement noire, par exemple, absorbe entièrement les rayons incidents. Un miroir les réfléchit presque entièrement. Une bouteille de bière réfléchit, diffuse, absorbe, transmet et disperse une certaine portion de la lumière incidente.

Comme le montre la figure 1.5, les rayons lumineux qui parviennent jusqu'à un observateur et qui participent donc à l'image :

- soit sont issus directement de la source lumineuse (A),
- soit ont été réfléchis par la surface d'un objet (C et E_1), diffusés (D), transmis ou dispersés.

D'autres rayons, bien sûr, ne participent pas à l'image (B, F qui est réfléchi puis absorbé et E_2 qui est transmis mais qui ne rencontre pas l'observateur).

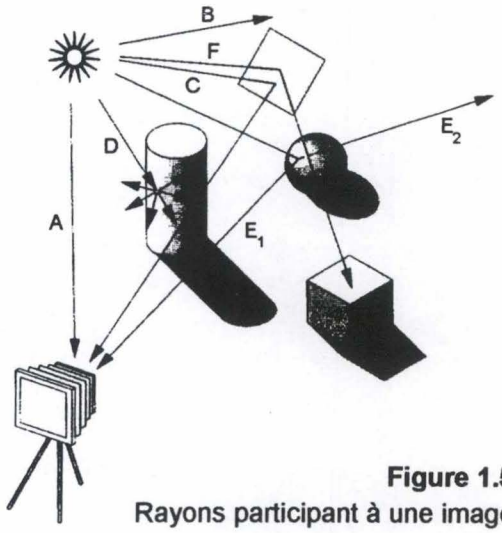


Figure 1.5
Rayons participant à une image

1.2.1.2. Le système visuel humain

La figure 1.6 schématise le processus de formation d'images tel qu'il est réalisé par un oeil humain.

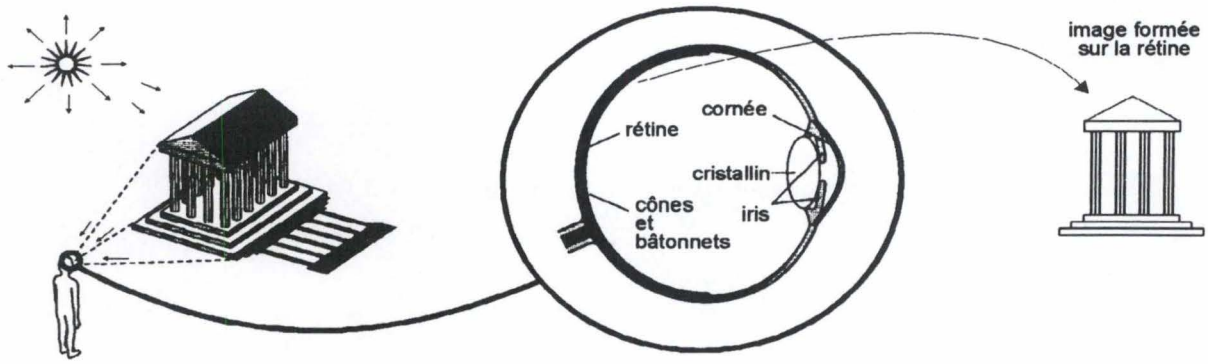


Figure 1.6 Processus de formation d'images avec le système visuel humain

Les rayons lumineux entrent dans l'oeil par la cornée, une structure transparente protectrice, puis par le cristallin. L'iris s'ouvre et se ferme pour ajuster la quantité de lumière entrante. Le cristallin forme une image sur la rétine, sur laquelle est étalée une couche de cônes et de bâtonnets. Ces capteurs réagissent quand ils sont frappés par un rayon de lumière. Leur taille, couplée avec les propriétés optiques du cristallin et de la cornée, déterminent la résolution (l'acuité visuelle) du système visuel humain. Les bâtonnets, contrairement aux cônes, sont plutôt sensibles à la lumière de faible intensité. Les trois types de cônes existants sont sensibles aux plages de longueurs d'ondes correspondant respectivement aux gammes de couleurs jaunes, vertes et bleues.

1.2.1.3. La caméra ponctuelle

La caméra ponctuelle est un modèle simple de caméra. C'est une boîte percée d'un petit trou sur le centre d'un de ses côtés ; un film est placé à l'intérieur, sur le côté opposé. Le trou est si petit qu'un seul rayon de lumière, à partir d'un point, peut y entrer. Par exemple, sur la figure 1.5, de tous les rayons diffusés résultant de l'interaction du rayon D avec le cylindre, un seul peut frapper le film de la caméra.

Comme le montre la figure 1.7, si on considère un système de coordonnées dont l'origine se trouve sur le trou de la caméra, et que celle-ci est orientée vers les z positifs, à tout point (x, y, z) duquel émane un rayon lumineux correspond, sur le film, un point (x_p, y_p, z_p) avec $z_p = d$. Le point (x_p, y_p, z_p) est la projection de (x, y, z) : la couleur du film à ce point sera la même que la couleur de (x, y, z). La figure 1.8 montre l'image qui est formée sur le film de notre caméra ponctuelle si on oriente celle-ci vers la façade avant d'un temple (comme celui de la figure 1.6).

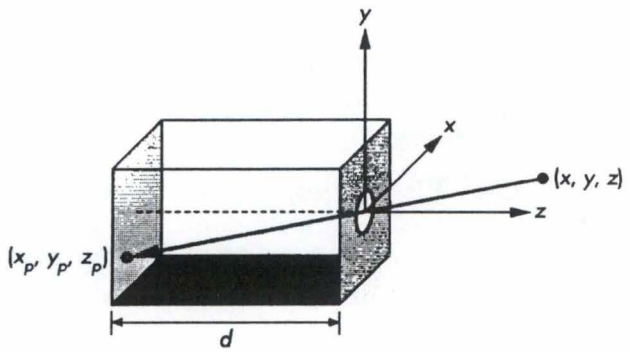


Figure 1.7 La caméra ponctuelle

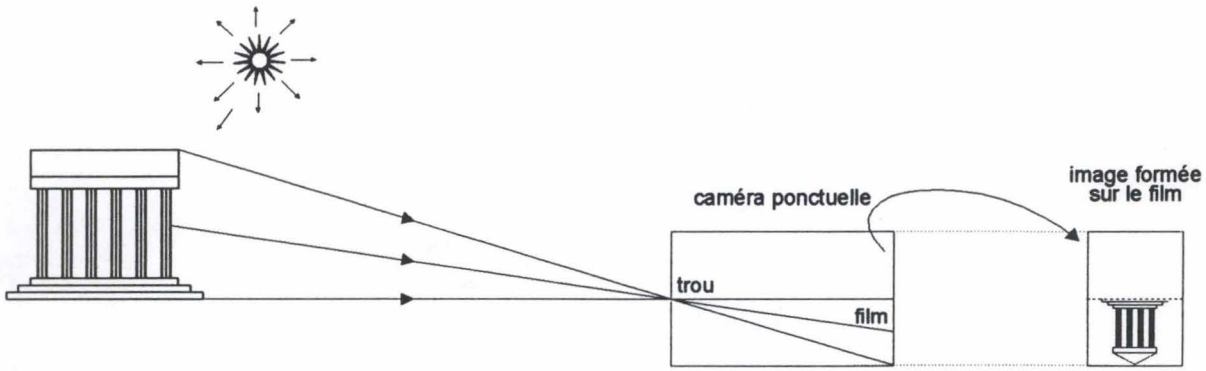


Figure 1.8 Processus de formation d'images avec la caméra ponctuelle

L'*angle de vue*, ou *champ* de la caméra (figure 1.9) détermine la plus large portion du monde que la caméra peut filmer. La caméra ponctuelle, puisqu'elle ne laisse passer qu'un seul rayon par point, a une *profondeur de champ* infinie : tout point filmé est net.

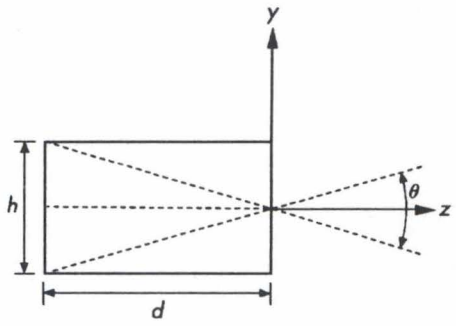


Figure 1.9 Angle de vue de la caméra ponctuelle

La caméra ponctuelle a deux désavantages. Premièrement, le trou est si petit que très peu de lumière entre dans la caméra. Deuxièmement, il est impossible d'ajuster la caméra pour avoir un angle de vue différent. On résout ces deux problèmes en remplaçant le trou par une lentille. Au plus celle-ci est large, au plus elle laisse passer de la lumière. En choisissant une lentille avec la longueur focale adéquate (ce qui, pour la caméra ponctuelle, est un choix équivalent à celui de la distance séparant le trou du film) on peut obtenir l'angle de vue souhaité (jusqu'à 180°). Les caméras dotées d'une lentille n'ont cependant qu'une profondeur de champ limitée.

1.2.2. Des systèmes optiques aux systèmes infographiques

Pour pouvoir simuler, à l'aide de l'outil informatique, le processus de formation d'images tel qu'il est réalisé par un système optique, il est nécessaire de :

1• disposer d'un modèle informatique de la couleur ;
2• modéliser en termes informatiques un observateur, une (ou des) source(s) lumineuse(s), les objets dont on veut obtenir des images, ainsi que les propriétés de leurs surfaces ;
3• mettre en oeuvre un processus qui, partant de ces modèles, forme des images sur l'output du système graphique.

Les APIs graphiques qui, comme OpenGL, sont basées sur le modèle de la caméra synthétique, fournissent au programmeur un ensemble de fonctions qui lui permettent de spécifier ces modèles et de contrôler le processus de formation d'images. Les détails d'implémentation, aussi bien software que hardware, lui sont cachés.

1.2.2.1. La couleur

Les trois types de cônes du système visuel humain sont sensibles aux plages de longueurs d'ondes

correspondant respectivement aux gammes de couleurs jaunes, vertes et bleues. Cette caractéristique de notre système visuel a entraîné la création d'un modèle à trois couleurs primaires. On peut utiliser ces trois couleurs pour approximer toutes les couleurs que l'on peut percevoir.

RGB (*Red - Green - Blue*) est l'application infographique de ce modèle à 3 couleurs. Dans ce modèle infographique, les bits servant à stocker un pixel du frame buffer sont divisés en trois groupes logiques, chacun représentant une valeur pour une couleur de base. Le standard actuel, pour représenter des images de qualité photographique, est d'utiliser 8 bits pour chaque gamme de couleur (rouge, verte et bleue), ce qui permet de représenter 2^{24} couleurs (± 16, 7 millions).

Beaucoup de systèmes ont cependant un frame buffer limité en profondeur, 8 bits par exemple. Diviser ces bits en petits groupes dont chacun représente une couleur de base est souvent irréalisable ou visuellement inacceptable. C'est la raison pour laquelle des systèmes à **couleur indexée** ont été créés.

Supposons que l'écran puisse afficher 2^{24} couleurs (2^8 nuances par couleur de base) mais que le frame buffer, dont la profondeur est limitée à 8 bits, ne puisse donc en spécifier que 2^8. L'idée est d'utiliser ces 2^8 valeurs non pas comme des valeurs de couleurs mais bien comme des index pointant vers une table de couleurs (figure 1.10), dont la taille, dans notre exemple, serait de 2^8 x 24 bits. Le programme utilisateur pourrait remplir les 2^8 lignes de cette table avec des valeurs pour les 3 couleurs de base, déterminant ainsi 256 couleurs dans une palette de 16,7 millions, résultat tout à fait acceptable pour beaucoup d'applications.

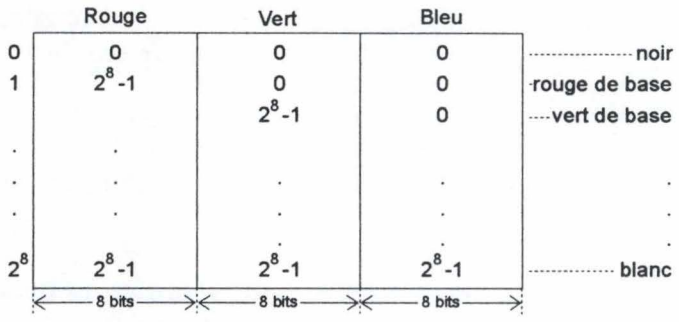


Figure 1.10 Table de couleurs

1.2.2.2. Modélisation des objets, des observateurs et des sources de lumière

Objets □ Les objets sont généralement définis comme des ensembles de sommets. Ces sommets déterminent des surfaces qui, à leur tour, déterminent des volumes. Les bibliothèques graphiques fournissent des objets de base : points, droites, polygones, et parfois, texte. Ces primitives sont souvent celles qui peuvent être affichées rapidement sur le hardware. Le code suivant définit un triangle dans openGL :

```
glBegin(GL_POLYGON);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(0.0, 0.0, 1.0);
glEnd();
```

En ajoutant, entre glBegin et glEnd, des sommets supplémentaires, on peut définir un polygone arbitraire. Si on modifie la valeur du paramètre de glBegin, on peut utiliser les mêmes sommets pour

définir une primitive géométrique différente. Par exemple, GL_LINE_STRIP utiliserait les sommets pour définir 2 segments de droite connectés, tandis que GL_POINTS les utiliserait pour définir 3 points.

Certaines APIs permettent à l'utilisateur de travailler directement dans le frame buffer en fournissant des fonctions qui lisent et qui écrivent des pixels. Certaines APIs permettent de définir des primitives plus complexes comme des courbes ou même des primitives 3D (sphères, cubes); au final, cependant, ces primitives sont presque toujours approximées par des primitives plus simples (polygones, droites). OpenGL permet d'accéder au frame buffer, de spécifier des courbes et des surfaces.

Les propriétés de la surface d'un objet sont les suivantes :

• un code RGB détermine la couleur des rayons diffusés par la surface quand celle-ci est éclairée par une lumière incidente parfaitement blanche. Pour spécifier ce code dans OpenGL, on fait appel à la fonction glColor3f avant de définir la surface :

```
. . .
glColor3f(0.0, 255.0, 0.0);      // couleur verte
. . .
glBegin(GL_POLYGON);
. . .
```

Un ensemble de codes RGB —une «texture»— peuvent être utilisés pour déterminer les couleurs d'une (ou plusieurs) surface(s) de l'objet. Cela se fait généralement en spécifiant un fichier *bitmap* ainsi qu'une manière d'«habiller» la surface de l'objet partant de ce fichier.

• Les propriétés optiques de la surface (la manière dont celle-ci diffuse, réfléchit, transmet et/ou disperse) sont généralement spécifiées par des vecteurs. Les systèmes graphiques les plus puissants utilisent RGBA, une extension de RGB qui permet de spécifier un facteur de transparence.

Observateur □ 4 paramètres sont nécessaires pour spécifier un observateur :
- sa position par rapport aux objets, qui généralement est déterminée par la position du centre de projection de sa lentille ;
- son orientation ;
- sa longueur focale, qui détermine son champ de vision ;
- la longueur et la largeur de son «film».

Sources de lumière □ Une source de lumière est définie par
- sa position,
- son orientation,
- la couleur des rayons lumineux qu'elle émet, leur intensité et leur directionnalité.

1.2.2.3. Processus de formation d'images

Les différentes étapes d'un processus possible de formation d'images découlant de l'application du modèle de la caméra synthétique sont illustrées par la figure 1.11.

Transformer □ La première étape consiste à réaliser une série de transformations entre systèmes de coordonnées, par exemple pour exprimer tous les sommets des objets dans un même système de coordonnées global, ou bien pour exprimer les objets dans le système de coordonnées local de l'observateur.

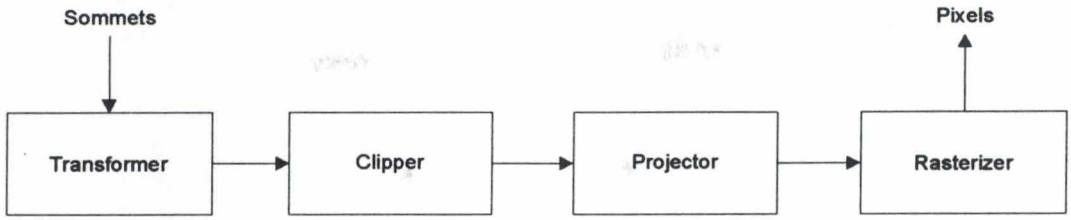


Figure 1.11 Processus de formation d'images infographiques

Clipper □ La deuxième étape consiste à déterminer le volume de l'espace virtuel qui entre dans le champ de l'observateur, sur base des spécifications de celui-ci : position, orientation, longueur focale et taille du film. Cette étape revient donc à identifier les objets — ou les parties d'objets— qui feront partie de l'image finale.

Projector □ Les rescapés de l'étape précédente sont, lors de la troisième étape, projetés sur un plan 2D, donnant naissance à l'image.

Rasterizer □ La quatrième et dernière étape convertit les données géométriques issues de l'étape précédente en pixels du frame buffer, qui sont alors affichés sur l'output du système graphique. La couleur de chaque pixel est déterminée par la combinaison de deux jeux de données :

- les données géométriques et optiques des objets ;
- les paramètres des sources lumineuses.

Cette technique de formation d'images est implémentée dans la plupart des bibliothèques graphiques modernes, comme PHIGS, GKS et OpenGL. Ces bibliothèques ne sont pas uniquement destinées à la production d'images 3D de haute qualité : tous les cas particuliers —images 2D, dessins, texte, icônes, etc— y sont traitables.

1.3. Applications de l'infographie

1.3.1. Affichage d'informations

Les techniques graphiques sont un moyen privilégié de communiquer de l'information. L'intérêt qu'elles ont toujours suscité, indépendamment des cultures et des époques, est lié au fait que notre système visuel est un outil puissant de traitement de l'information et de reconnaisance de formes. *La perception humaine est hautement tolérante aux bruits. Les défauts d'une image ou d'un son n'affectent guère notre compréhension du contenu, alors que l'identification et la suppression des bruits est un problème de taille en Intelligence Artificielle.* [...] *La reconnaissance de formes est un processus remarquable : l'apparence des objets change constamment sans que nous éprouvions de difficulté à leur donner un sens. Même face à un stimulus que nous n'avons jamais rencontré, nous n'éprouvons aucune difficulté à le catégoriser.* [...] *La vision est probablement le sens qui a fait l'objet du plus de travaux aussi bien en psychologie cognitive qu'en Intelligence Artificielle* [deK95].

L'affichage d'informations constitue donc un type d'application privilégié de l'infographie. Dans le domaine scientifique, par exemple, l'infographie aide les chercheurs à interpréter l'énorme quantité de données qu'ils génèrent. Les statisticiens disposent d'outils graphiques leur permettant d'appréhender visuellement les ensembles de données qu'ils manipulent. L'imagerie médicale

regroupe un nombre de plus en plus impressionnant de techniques d'analyse offrant aux médecins un support leur permettant d'établir des diagnostics plus rapidement, ou de manière plus précise.

1.3.2. Conception

Ingénieurs, architectes, informaticiens, artistes... Tous les concepteurs modernes ont découvert dans l'infographie un moyen de développer leurs réalisations.

Les logiciels de CAO (Conception Assistée par Ordinateur), pour les deux premiers, permettent notamment de visualiser les créations à différents niveaux de détail ou à différents états d'avancement.

Les outils CASE (*Computer-Aided Software Engineering*), pour les troisièmes, qui sont de plus en plus utilisés dans les processus de développement de logiciels, sont souvent les implémentations de langages graphiques développés pour le pouvoir d'expression qu'ils peuvent potentiellement fournir.

Les programmes de création graphique offrent aux derniers un large éventail de possibilités concernant la génération d'images, fixes ou animées.

On retrouve deux grands types de logiciels de conception graphique :

- les logiciels de modélisation permettent de définir les modèles d'objets, d'observateurs et de sources lumineuses. Ils permettent aussi de définir les mouvements, ou animations, qui doivent être réalisées par ces modèles. Il existe plusieurs manières de définir ces animations; nous ne les avons pas abordées dans ce chapitre. Nous parlerons à partir du chapitre 3 de celle qui consiste à enregistrer les mouvements d'un être humain pour les appliquer à un personnage 3D. Les autres techniques sortent du cadre de ce travail.
- Les logiciels de rendu, sur base des modèles générés par les logiciels de modélisation, calculent les images finales.

L'utilisateur de ces logiciels voit donc le processus de formation d'images comme un processus en deux étapes : la modélisation et le rendu.

1.3.3. Simulation

Depuis que les systèmes graphiques sont capables de générer des images haute définition à un rythme supérieur aux 22 images par seconde traitables par l'oeil humain, les chercheurs et les ingénieurs les utilisent comme simulateurs. Le domaine de l'aviation a été l'un des premiers à bénéficier de ce type d'application pour l'entraînement des pilotes. Celui de la médecine s'est ensuite équipé de systèmes permettant aux étudiants en chirurgie de s'exercer sur des patients inexistants.

L'imitation du réel est devenue une mode. Images photoréalistes, jeux vidéo, images de synthèse au cinéma ou à la télévision... le virtuel, interactif ou non, trompe de plus en plus fréquemment.

1.3.4. Interfaces hommes-machines

Notre interaction avec les ordinateurs est dominée par un paradigme graphique: fenêtres, menus, icônes... autant d'objets interactifs que l'on retrouve dans les interfaces des systèmes d'exploitation actuels.

Chapitre 2
Neurones Animation SA

Le projet GabbyCapture s'est déroulé dans la société liégeoise Neurones Animation SA. L'objectif de ce chapitre est de présenter cette société.

Neurones Animation SA fait partie du groupe Neurones. Nous allons donc d'abord présenter celui-ci (2.1). Après l'avoir introduit par un petit historique (2.1.1), nous aborderons sa structure (2.1.2) et ses activités (2.1.3).

Nous présenterons ensuite Neurones Animation SA (2.2). Après avoir exposé sa structure (2.2.1) et ses activités (2.2.2), nous présenterons ses projets futurs (2.2.3). Nous ne parlerons pas, dans ce chapitre, du projet GabbyCapture puisqu'il fera l'objet des chapitres 3 et suivants.

2.1. Le groupe Neurones

2.1.1. Introduction

En 1989, Paul Hannequart décide d'abandonner sa carrière de médecin pour fonder Neurones Cartoons. Cette petite société liégeoise est constituée d'un noyau d'informaticiens et a pour objectif initial de créer des logiciels d'aide à la conception de séries d'animation [1]. Très vite, elle se rend compte

[1] Une série d'animation est une série télévisée dont les images sont produites grâce à une technique d'animation particulière.

• La technique d'animation la plus répandue est celle du dessin animé. Elle consiste à filmer des séquences de dessins ; 24 dessins sont nécessaires pour produire une seconde de film. Un dessin animé peut être réalisé par ordinateur, du moins partiellement : en général, les dessins sont exécutés à la main, puis sont scannés et coloriés par ordinateur. Celui-ci les fait ensuite défiler à un rythme de 24 par seconde. Les séries d'animation réalisées, avec ou sans ordinateur, grâce à la technique du dessin animé sont appelées *séries d'animation 2D* (ou *dessins animés*) puisqu'elles sont composées de dessins qui ont été exécutés en deux dimensions. →

de l'intérêt que peuvent présenter les techniques 3D (c'est-à-dire celles qui sont issues du modèle de la caméra synthétique) pour produire des images par ordinateur. Deux outils sont alors développés: Chromos, un logiciel de modélisation d'objets 3D, et Kiss, un logiciel de *synchronisation labiale*. Celui-ci consiste à animer en temps réel la bouche d'un personnage en images de synthèse à partir d'une séquence vocale captée par micro. Commercialisé et distribué par Softimage, une filiale de Microsoft, Kiss compte parmi sa clientèle des noms comme Sega et Walt Disney.

Figure 2.1
Le concombre masqué

Parallèlement à ses activités de conception de programmes, Neurones commence à produire et à réaliser des dessins animés et des séries d'animation 3D. Parmi celles-ci, on trouve notamment l'épisode-pilote du *Concombre Masqué* (figure 2.1), qui lui a valu le prix du public au FIFOM de Montréal en 1991, et la série *Insektors*, coproduite en 1993 avec Fantôme, France 3 et Canal+, qui a reçu un *Emmy Award* à New York en 1994. Côté dessins animés classiques, la production atteint en 1995 le rythme de 3 à 4 séries de 26 épisodes de 26 minutes [1] par an. Les contrats d'association à la production pour l'année 1996 sont, entre autres, signés avec Gaumont pour la série *Dragon Flyz*, les Films de la Perrine pour *SOS, Bout du monde*, France Animation pour *Sanbarbe*... Un logiciel interne, Trag, est créé pour couvrir certaines parties du processus de réalisation de ces séries.

Le concept de «cyberacteur» naît en 1994. L'idée est d'animer en temps réel un petit être de synthèse, une «marionnette virtuelle» 3D que l'on fait parler et à laquelle on donne des expressions et des mouvements. Un système ad hoc est développé à partir de la même année ; son composant principal est Gabby, un logiciel d'animation qui intègre notamment la synchronisation labiale de Kiss. Des cyberacteurs personnalisés sont vendus à Coca-Cola, Toshiba, Ricard, Belgacom et bien d'autres (celui de Belgacom est présenté à la figure 2.2). Ils sont utilisés lors de foires, de salons ou de festivals, pour communiquer avec des clients potentiels d'une manière attrayante et originale.

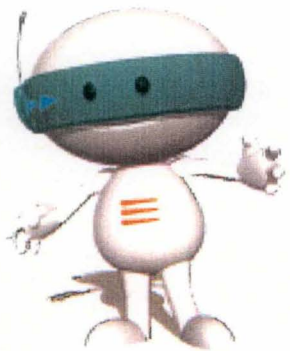


Figure 2.2 Digit,
le cyberacteur de Belgacom

Aujourd'hui, le groupe Neurones emploie plus de 350 personnes dans 7 sites répartis en Belgique, en France, au Grand-Duché du Luxembourg, en Corée et au Canada. Son chiffre d'affaires pour l'année 1997 est de 360 millions BEF et devrait atteindre plus de 600 millions en 1998. La société d'ancrage liégeois compte maintenant parmi les firmes européennes les plus importantes sur les marchés de l'animation 2D et 3D.

← • La technique de l'animation en images de synthèse se base sur le modèle de la caméra synthétique : elle consiste à modéliser, à l'aide d'un ordinateur, des personnages et des décors en 3 dimensions, puis de les «filmer» dans différentes positions, avec différents angles de vue. 24 images sont nécessaires pour produire une seconde de film. Les séries d'animation réalisées grâce à la technique de l'animation en images de synthèse sont appelées *séries d'animation 3D* (ou *séries d'animation en images de synthèse*) : elles sont composées d'images obtenues à partir de modèles tridimensionnels.

• D'autres techniques existent, comme celle qui consiste à filmer des figurines (en plasticine, en fil de fer...) dans un décor miniature.

[1] Plusieurs standards de diffusion existent : 26x26, 13x5, 10x5 minutes, etc.

Le trait commun des différentes activités de Neurones est la réalisation d'images animées par ordinateur. Si nous devions classer ces activités selon les types d'applications de l'infographie présentés dans le chapitre 1, nous dirions qu'elles se réclament à la fois de la conception (artistique) et de la simulation.

En effet, dans la majorité des cas, les scénaristes et réalisateurs de dessins animés —Neurones faisant partie des seconds— s'inspirent du réel, l'imitent dans une certaine mesure et l'intègrent dans un monde imaginaire. Jean Image, un des pionniers français du métier, illustre bien ce trait fondamental quand il évoque la manière d'inventer un nouveau personnage animé : [...] *il faut en général s'inspirer de la vie réelle et le styliser au maximum, sans pour autant lui retirer sa personnalité* [Ima].

La série d'animation 3D met à profit toute la puissance du modèle de la caméra synthétique. Si cette particularité rend la série 3D a priori plus apte à la simulation, elle ne l'éloigne pas pour autant de sa nature de fiction et lui permet, l'incite, même, à dépasser les limites du monde réel par la modélisation d'objets ou de personnages tout à fait fantaisistes.

En ce qui concerne les cyberacteurs, le constat est identique. Leur physionomie, leurs expressions et leurs mouvements ont été conçus non pas dans le but d'imiter parfaitement ceux des humains, mais bien dans celui de s'en inspirer pour produire un effet original.

2.1.2. *Structure*

Les sociétés qui font partie du groupe Neurones sont présentées à la figure 2.3.

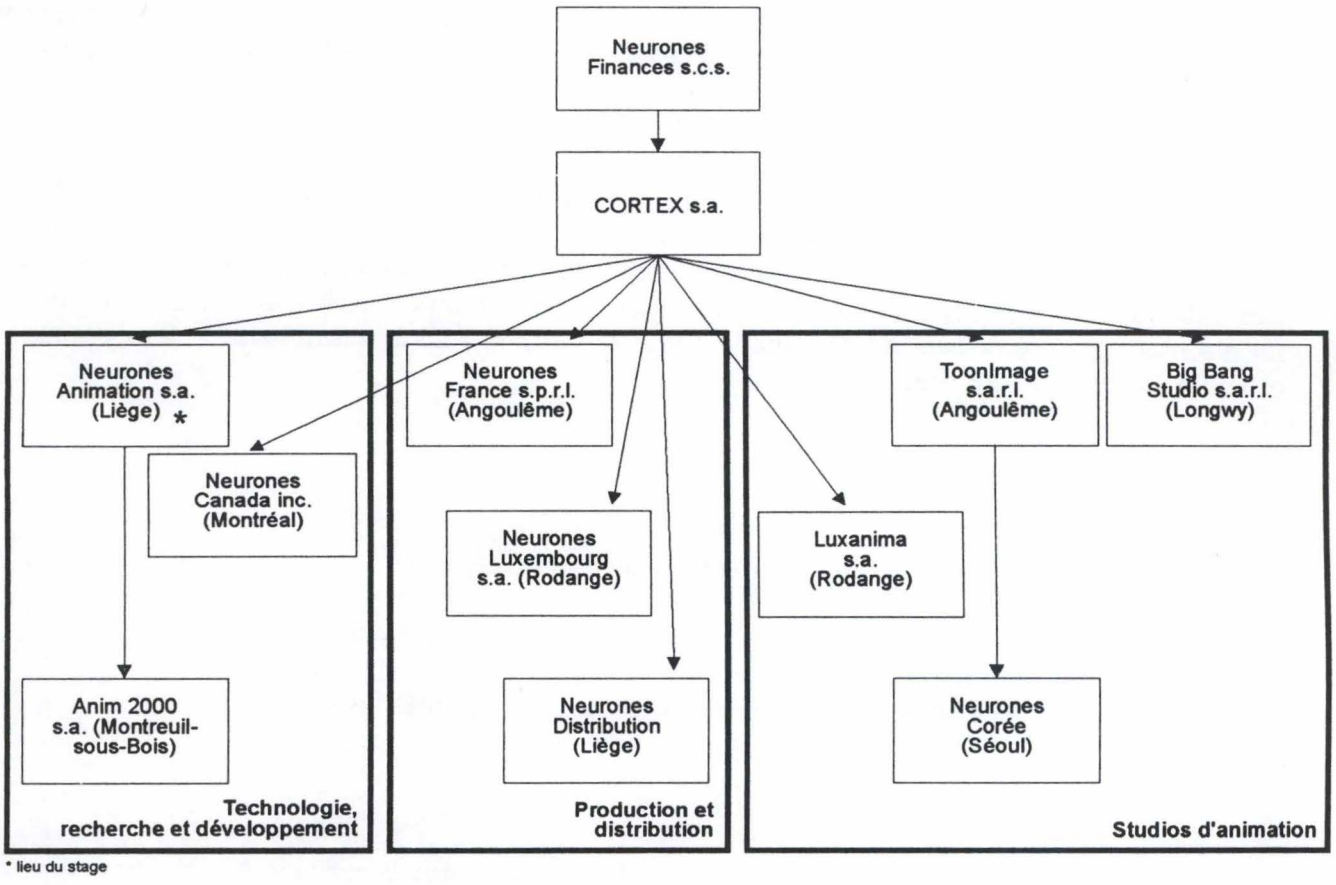


Figure 2.3 La structure du groupe Neurones

Le holding CORTEX s.a. appartient à une société en commandite simple, Neurones Finances s.c.s. Celle-ci est détenue par Paul Hannequart (±37%), Marc Minjauw (±37%) et, depuis mars '98, le holding public de financement Meusinvest (±25%).

Luxanima, ToonImage et Big Bang Studio sont tous trois des studios d'animation entièrement informatisés, qui emploient respectivement 60, 50 et 16 personnes. Ils prennent en charge la partie «informatique» du processus de réalisation des séries d'animation 2D ; les quelque 200 dessinateurs de Neurones Corée, quant à eux, effectuent la partie «papier» de ce processus.

Neurones Animation s.a. est le centre technologique du groupe. Fidèle à sa vocation initiale, l'entreprise liégeoise s'occupe du développement et de la maintenance de logiciels, dont la plupart est utilisée uniquement en interne, comme Trag pour le dessin animé et Gabby pour les cyberacteurs. Elle traite également l'ensemble de la chaîne de production de ces derniers et constitue un studio d'appoint pour la réalisation de séries en images de synthèse. Elle comprend 24 personnes, dont Paul Hannequart.

Anim 2000 est née dans le but de développer et de commercialiser un logiciel de *story-boarding*[1]. Elle est composée de programmeurs de chez Neurones Liège à 50%, d'employés de l'INA (l'Institut National de l'Audiovisuel en France) à 25% et d'employés de France Animation pour les 25% restants.

Le rôle des deux personnes travaillant à Neurones Canada est de promouvoir outre-mer la technologie des cyberacteurs, en essayant bien sûr de décrocher quelques contrats de vente.

Neurones Luxembourg et Neurones France sont des sociétés de production audiovisuelle qui co-financent les séries d'animation réalisées par Neurones. Neurones Distribution, nouvellement créée, est chargée de vendre aux télévisions du Bénélux le catalogue actuel des dessins animés de Neurones, qui compte ± 75 heures de programmes. Elle est aussi chargée de distribuer les productions de ses clients, en contrepartie de quoi elle participe à leur plan de financement.

Les fonctions des différentes sociétés du groupe montrent l'importance relative que celui-ci accorde à chaque domaine d'activité dans lequel il est actif. Le dessin animé est largement en tête : 4 studios, 2 entreprises de production et une société de distribution lui sont entièrement consacrés. Les cyberacteurs suscitent un intérêt plus mitigé ; quant aux séries d'animation 3D, elles ne représentent qu'une activité occasionnelle. Cette répartition reflète naturellement les opportunités qui sont offertes par les marchés correspondants.

2.1.3. *Activités*

Les trois grandes activités du groupe Neurones sont :

- la production, la réalisation et la distribution de séries d'animation 2D et 3D,
- la réalisation de cyberacteurs,
- la recherche et le développement, activité qui supporte les deux premières.

[1] Le story-boarding est une étape majeure de la pré-production d'une série d'animation. Nous verrons exactement de quoi il s'agit dans le point 2.1.3.1.

Nous ne parlerons ici que de la première activité (2.1.3.1). Nous aborderons les deux autres quand nous parlerons de la société liégeoise (2.2.2.1 et 2.2.2.2), puisqu'elles y sont presque exclusivement réalisées.

2.1.3.1. Production, réalisation et distribution de séries d'animation

La réalisation de séries 3D étant pour le moins anecdotique chez Neurones —du moins dans les dernières années—, nous avons choisi de la laisser de côté. Nous ne nous préoccuperons ici que des des séries d'animation 2D.

La figure 2.4 expose les différentes étapes du processus de réalisation d'un dessin animé tel qu'il est effectué chez Neurones.

Scénario, story-board et model-sheets □ Le point de départ de toute série animée est l'écriture d'un scénario. Sur base de ce document purement textuel, il s'agit de présenter visuellement, plan par plan, la manière dont l'action va se dérouler : c'est ce qui est fait dans le story-board. Un plan est une unité élémentaire de film, un enchaînement d'actions qui ont lieu dans le même décor. Le story-board contient, pour chaque plan :

- un ou plusieurs croquis ;
- des commentaires textuels concernant les dialogues (ou les voix-off), les bruitages et la musique, les animations (personnages, objets, ...) et les mouvements de caméra [1] ;
- la durée du plan et le timing des animations.

Les model-sheets sont les croquis des personnages et des objets à animer, vus sous différents angles, dans plusieurs positions, avec différentes expressions. Ils sont, avec le scénario et story-board, les résultats de la phase de pré-production, ou de spécification du travail à réaliser.

Layout □ La production à proprement parler commence avec le layout. Les documents reçus en aval sont interprétés pour produire des instructions précises, plan par plan, image par image.

- Le *layout décor* est une première mise en place au crayon des différents éléments du décor ;
- le *layout cadres* définit les mouvements de la caméra et du décor ;
- le *layout animation* contient les dessins nécessaires pour décrire l'action des personnages dans chaque plan : dessins de début et de fin de plan, positions

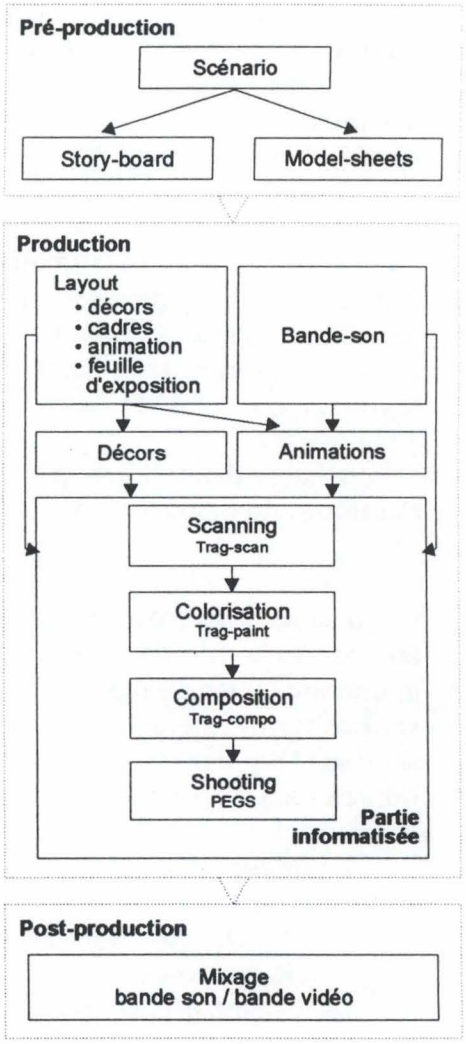


Figure 2.4
La réalisation d'un dessin animé

[1] En effet, le principe du dessin animé est de filmer, que ce soit avec une caméra réelle ou à l'aide d'un équivalent informatique, 24 dessins pour obtenir une seconde de film. Pour ce faire, on dessine généralement un décor fixe, et on lui superpose des dessins que l'on reproduit avec des décalages progressifs, ce qui produit une impression de mouvement.

intermédiaires, expressions importantes.

- la *feuille de prise de vues / d'exposition* comprend les informations de timing du plan : animations des personnages, déplacements de caméra, synchronisation précise avec les éléments sonores. Elle se présente souvent comme un tableau dont chaque ligne dénote une image du film. Chaque item de colonne correspond à un paramètre de cette image : un dessin faisant partie d'une animation, la position de la caméra, la position du décor, etc.

Enregistrement de la bande-son □ L'enregistrement des paroles, des bruitages et de la musique se fait en parallèle avec la réalisation du layout. Cependant, l'enregistrement des dialogues doit se faire avant la réalisation des animations afin de pouvoir dessiner les mouvements de bouche adéquats.

Réalisation des décors et des animations □ Le layout décor sert de base pour réaliser le dessin des décors définitifs. Le layout animation et le timing de la feuille de prise de vue permettent de réaliser les dessins-clé des mouvements des personnages —c'est le rôle de l'animateur— et d'autre part de fournir les dessins intermédiaires nécessaires pour compléter à 24 dessins par seconde —c'est le travail de l'intervalliste.

Etapes informatiques □ L'ensemble des décors et des séquences de crayonnés d'animation sont scannés, puis mis en couleur. Les plans sont ensuite composés et calculés, image par image, sur base du layout cadres et de la feuille de prise de vues : le décor est positionné et superposé aux dessins d'animation, la caméra (virtuelle dans notre cas) est cadrée, etc. La séquence qui en résulte est transférée sur support magnétique (shooting): bande vidéo au format NTSC, PAL,... Les modules de Trag —Trag-Scan, Trag-Paint, Trag-Compo... — sont les outils qui supportent ces activités (mis à part le shooting, qui est réalisé avec le logiciel commercial PEGS).

Mixage des bandes son et vidéo □ Le mixage son et vidéo s'effectue à l'aide des techniques classiques de manipulation de bandes magnétiques.

Neurones n'intervient dans cette chaîne de production qu'après le layout et la création de la bande-son. Ce sont des sociétés comme France Animation ou Canal+ qui se chargent de ces premières étapes. Au terme de celles-ci, Neurones Corée est susceptible d'entrer en scène pour réaliser les animations et les décors. Le tout peut ensuite être envoyé soit à Luxanima, soit à ToonImage, les deux studios étant entièrement équipés pour couvrir les étapes informatiques, depuis le scanning jusqu'au shooting. Big Bang studio réalise une partie de la colorisation des dessins animés qui aboutissent à Luxanima. La post-production (le mixage des bandes son et vidéo) n'est pas du ressort de Neurones.

Le financement des dessins animés est assuré par Neurones France et Neurones Luxembourg, en partenariat avec d'autres maisons de production.

- *SOS Bout du Monde*, coproduite avec les Films de la Perrine, Cinegroupe Canada et Ravensburger, a été diffusée sur France 2 lors du dernier trimestre 1997.
- *SanBarbe*, coproduite avec France Animation et France 3, a été diffusée sur France 3 en '97.
- *Les Pirates de Noël*, en coproduction avec Antefilms, l'INA et France 3, a été diffusée les 24 et 25 décembre sur Canal+.
- *Patrouille 03*, en coproduction avec France Animation, sera diffusée cette année.
- *Petit Potam*, en coproduction avec Marina Production, France 3, ZDF et Eva Entertainment, sera également diffusée en 1998.

Depuis janvier 1998, avec la création de Neurones Distribution, le groupe s'est doté d'un nouveau service : la distribution de dessins animés, qu'il s'agisse des siens ou de ceux d'autres producteurs.

2.2. Neurones Animation SA

2.2.1. *Structure*

L'organigramme de Neurones Animation SA, tel qu'il se présentait au 31 janvier 1998, fait l'objet de la figure 2.5.

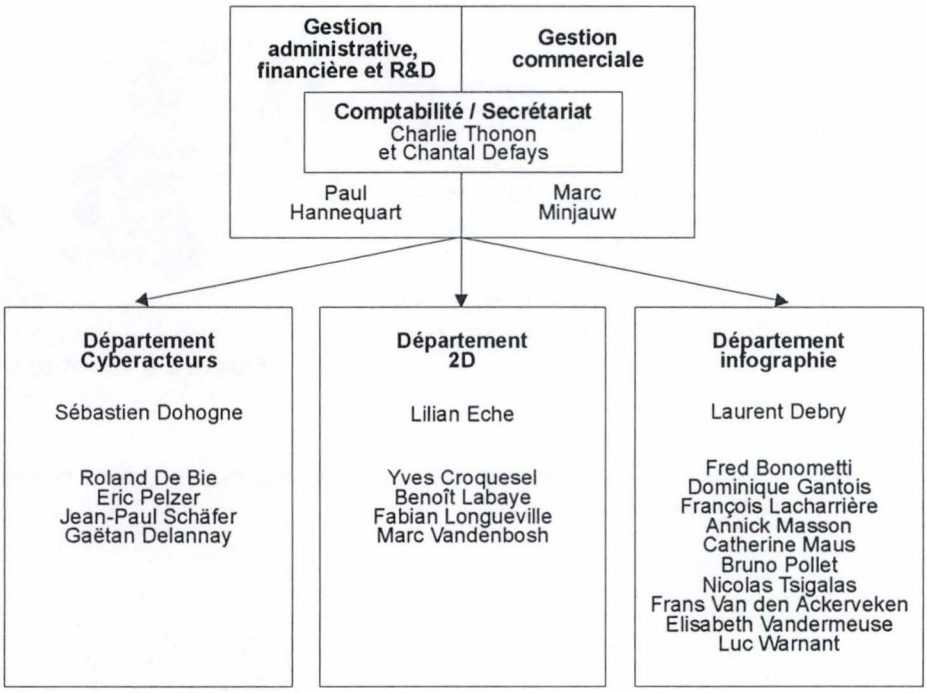


Figure 2.5 La structure de Neurones Animation SA

Le département cyberacteurs prend en charge l'activité de recherche et de développement qui concerne les cyberacteurs, c'est-à-dire, comme nous allons le voir dans la section suivante, le développement et la maintenance du logiciel Gabby. Ce département joue également un rôle dans la chaîne de production des cyberacteurs.

Le département 2D effectue l'activité de recherche et de développement qui a trait à la réalisation de séries d'animation (2D), c'est-à-dire le développement et la maintenance du logiciel Trag et du logiciel de story-boarding.

Le département infographie prend en charge la plus grosse partie de la chaîne de production des cyberacteurs. Il s'occupe aussi, occasionnellement, de la réalisation de séries d'animation 3D.

2.2.2. *Activités*

2.2.2.1. Réalisation de cyberacteurs

Cette section aborde le fonctionnement des marionnettes virtuelles (dont quelques spécimens sont présentés à la figure 2.6) et l'activité qu'elles entraînent dans la société qui les a inventées.

Le principe est simple. Une télévision est installée, par exemple, sur le stand d'une entreprise dans une foire commerciale. Un cyberacteur y est affiché. Une caméra filme les gens qui regardent la télévision. Un animateur humain «tire les ficelles» de la marionnette dans un poste de pilotage invisible du public:

- il voit celui-ci grâce à la caméra ;
- il parle dans un micro, ce qui a pour effet de faire bouger en temps réel les lèvres du cyberacteur, et de diffuser le son de sa voix — filtrée ou non — dans les enceintes acoustiques de la télévision ;
- il manipule une tablette graphique pour insuffler en temps réel des postures, des expressions faciales et des mouvements au cyberacteur.

Figure 2.6 Des cyberacteurs

L'ensemble matériel et logiciel de Neurones qui met en oeuvre ces idées porte le nom de Gabby Event et fait l'objet de la figure 2.7.

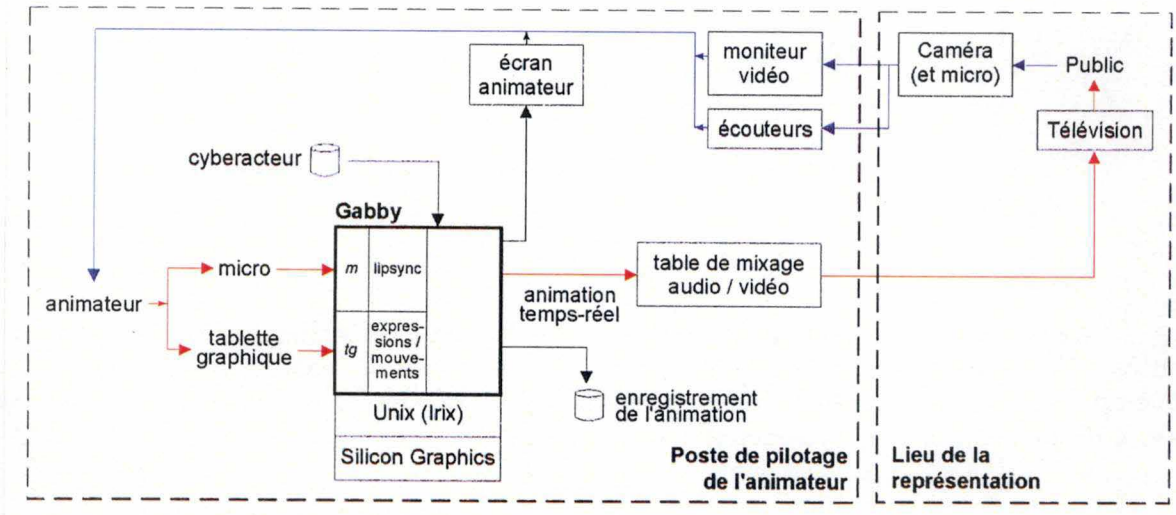


Figure 2.7 Gabby Event

Le composant principal du système est le logiciel Gabby, qui tourne sur une plateforme Unix (Silicon Graphics). Une vingtaine de fois par seconde, l'image du cyberacteur chargé dans Gabby, conditionnée par les inputs du micro et de la tablette, est affichée sur l'écran de l'animateur —en guise de feedback— et est envoyée, avec le son de la voix de l'animateur, sur la table de mixage audio/vidéo. Le résultat, après avoir subi quelques réglages, apparaît et se fait entendre sur la télévision que le

public est en train de regarder. Dans Gabby, les données du micro sont récupérées grâce à un pilote de périphérique (*m* sur la figure 2.7) et utilisées par un algorithme de synchronisation labiale (dénoté lipsync sur la même figure) pour calculer le mouvement des lèvres du cyberacteur. Les données de la tablette, récupérées par un pilote (*tg* sur la figure 2.7), sont manipulées par un module responsable des mouvements et des expressions du cyberacteur. Gabby offre la possibilité d'enregistrer l'animation.

Un premier flux de données (en rouge sur la figure 2.7) assure donc la communication de l'animateur vers le public. Un second flux, allant dans le sens inverse (en bleu sur la figure 2.7), réalise la dimension interactive du système. Les images de la caméra sont transmises sur le moniteur vidéo dans le poste de pilotage de l'animateur et le son aboutit dans les écouteurs de celui-ci. La figure 2.8 montre un croquis de ce poste de pilotage.

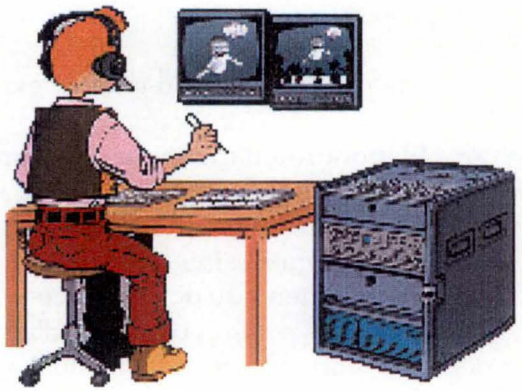


Figure 2.8 Croquis du poste de pilotage

Revenons plus en détail sur la tablette graphique. Comme l'illustre la figure 2.9, cette tablette est divisée en cases logiques. L'interaction s'effectue au moyen d'un stylet optique que l'on appuie ou que l'on fait glisser sur ces cases. Nous n'allons pas détailler les fonctionnalités de chacune d'entre elles, d'autant plus que certaines sont spécifiques à un cyberacteur particulier.

Sur la figure 2.9, la case en dessous à gauche permet de déplacer le personnage d'avant en arrière et de gauche à droite dans son espace virtuel. Les deux petites rangées du coin supérieur droit de la tablette fonctionnent comme des interrupteurs : les actionner a pour effet de doter le cyberacteur d'un accessoire particulier (bonnet, chapeau, pipe, lunettes, etc.). Déplacer le stylet sur une des autres cases du haut fait varier la proportion de l'expression qui lui correspond, et modifie d'autant l'émotion générale du personnage. Celui-ci est donc caractérisé par une «charte émotionnelle», c'est-à-dire un ensemble d'expressions-clé dont la gradation et la combinaison déterminent l'expressivité du personnage. La figure 2.10 montre un exemple de charte.

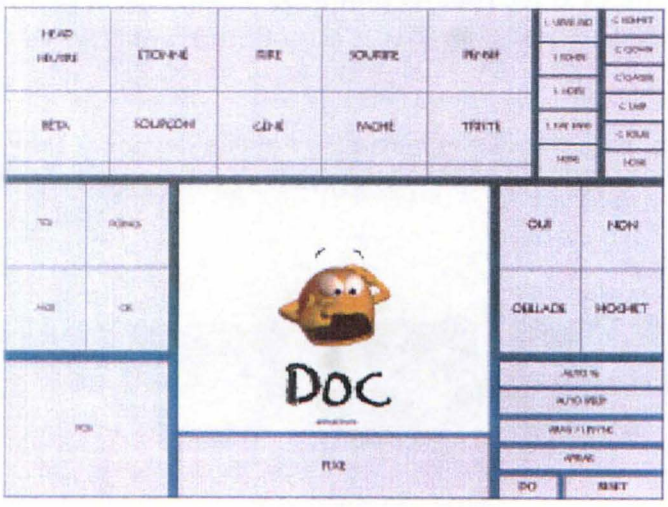


Figure 2.9 La tablette graphique de Doc

Certains mouvements ou mimiques, plus fantaisistes ou plus complexes, ne peuvent pas être obtenus par le jeu des expressions graduées. Ils se retrouvent donc en tant qu'animations pré-enregistrées («serrer les poings», «faire un clin d'oeil», ...) que l'on peut déclencher en appuyant sur les cases situées de part et d'autre de la case centrale (toujours sur la figure 2.9). Ces cases «court-circuitent» légèrement l'aspect temps-réel de Gabby, puisque l'animateur perd tout contrôle sur le cyberacteur pendant que ce type d'animation se déroule.

La tablette graphique est relativement bien adaptée pour générer des expressions faciales. Par contre, elle n'est pas pratique

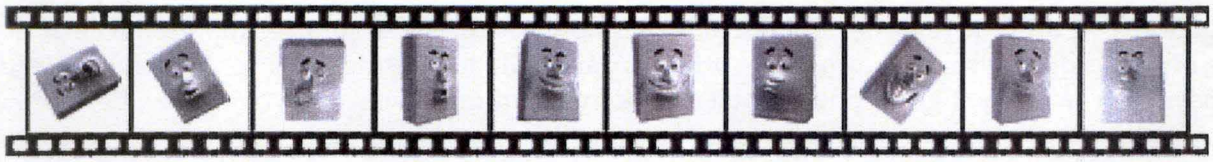


Figure 2.10 La charte émotionnelle de Glucoz

pour insuffler des déplacements, et encore moins des mouvements corporels. Beaucoup de cyberacteurs n'ont d'ailleurs qu'une tête, un buste et des bras, et cela est également dû aux difficultés que Gabby Event peut rencontrer dès lors qu'il s'agit de faire le rendu en temps réel de modèles 3D plus complexes.

Les paragraphes qui suivent expliquent la manière dont les cyberacteurs sont réalisés.

Elaboration d'un dossier de commande □ La spécification du personnage, des expressions qu'il devra prendre, des animations qu'il devra effectuer, des décors, des objets à manipuler et des variantes vestimentaires, ainsi que les model sheets correspondants, forment le contenu du dossier de commande. Cette commande peut consister à réaliser le cyberacteur d'un personnage existant (logo ou mascotte d'une entreprise) ou non. La figure 2.11 présente quelques model-sheets d'un cyberacteur, ainsi que la mascotte sur base de laquelle ils ont été réalisés — la mascotte du magasin de jouets Maxi-Toys.

Figure 2.11
La mascotte de Maxi-Toys et quelques model-sheets

Modélisation □ Pour modéliser la tête du cyberacteur, une petite sculpture en papier mâché est d'abord façonnée sur base des model sheets. Cette sculpture sert de point de départ pour modéliser la tête à l'aide d'un outil informatique, Hash Animation. On fournit au modèle ainsi créé différentes caractéristiques physiques de base, appelées *shapes*, définissant comment et dans quelles limites vont pouvoir bouger les sourcils, la bouche, les yeux,etc. On crée également différentes expressions que l'on obtient par combinaison de shapes. On définit ensuite la couleur et les textures que l'on va appliquer aux différentes parties du visage. La figure 2.12 reprend, pour le cyberacteur de Maxi-Toys, la sculpture, le modèle en fil de fer et trois modèles ayant chacun une position de bouche différente, c'est-à-dire ayant une valeur différente pour la shape de la bouche.

Le **corps**, les **objets** et les **décors** sont modélisés, coloriés et texturés sur base des model sheets à l'aide de logiciels comme Softimage ou 3D Studio MAX, deux des logiciels d'images de synthèse

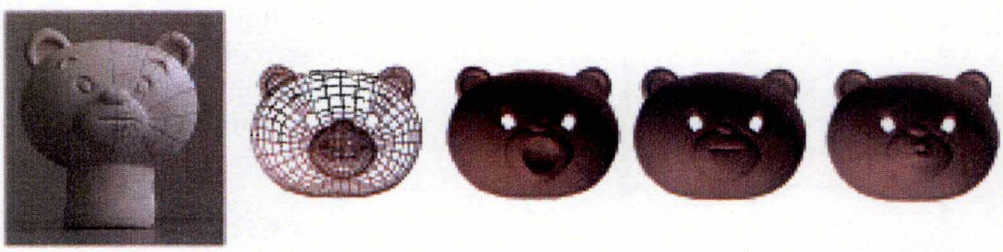


Figure 2.12 Modélisation du cyberacteur de Maxi-Toys

(modélisation et rendu) les plus répandus.

Réalisation des animations et attitudes pré-enregistrées □ Le travail pré-enregistré est également réalisé avec Softimage et 3D Studio MAX.

Intégration □ Les softs commerciaux précédemment cités sont des outils de modélisation aussi bien que des outils de rendu; cependant, aucun d'entre eux n'offre la possibilité de faire un rendu temps réel, ou du moins pas de la manière dont Gabby le réalise. La dernière étape de la chaîne consiste donc à intégrer les modèles et les animations dans Gabby. Cette étape comprend :

- l'importation, l'adaptation et la mise en commun des fichiers de modélisation et d'animation;
- la conception des cases logiques de la tablette graphique correspondant aux attitudes, émotions et mouvements du cyberacteur.

La chaîne de production des cyberacteurs s'effectue entièrement à Liège. Neurones Canada, quant à lui, prend en charge les aspects commerciaux pour la région des USA et du Canada: la recherche de nouveaux clients, la vente de cyberacteurs, les démonstrations... En Europe, Neurones Animation Liège s'est délesté de ces tâches en créant une formule de «franchisés Gabby» : chaque société ayant ce statut touche, en contrepartie, un bénéfice sur la vente du produit.

Que signifie exactement le «produit vendu»? Le client achète le droit exclusif d'utiliser la marionnette ; un système Gabby Event lui est loué à chaque fois qu'il désire faire une représentation (on lui fournit, au besoin, un animateur). Les systèmes Gabby Event restent la propriété de Neurones.

2.2.2.2. Recherche et développement

L'activité de recherche et développement est entièrement réalisée à Neurones Animation Liège. Sa philosophie est la suivante :

- l'objectif principal est de fournir un support aux activités de réalisation de dessins animés et de cyberacteurs ;
- l'objectif secondaire est de commercialiser certains logiciels informatiques afin de fournir une rentrée d'argent supplémentaire au groupe.

L'objectif principal est concrétisé par le développement et la maintenance du logiciel Trag pour le dessin animé et Gabby pour les cyberacteurs. Ces logiciels «développés maison» le sont

- soit parce qu'il est moins intéressant, financièrement parlant, d'acheter des licences de logiciels (c'est le cas du logiciel PEGS qui est progressivement remplacé dans les studios de Neurones par Trag) ;
- soit parce que le logiciel nécessaire n'existe pas dans le commerce (Gabby).

La commercialisation de Kiss a répondu à l'objectif secondaire de la recherche et du développement ; à l'heure actuelle, c'est le développement d'un logiciel de story-boarding qui rentre dans cette optique. A terme, ce soft devrait couvrir toutes les étapes de la pré-production d'un dessin animé, jusqu'au layout. Il sera d'abord vendu aux collaborateurs de Neurones (France Animation en tête) et dans le commerce par la suite.

2.2.3. Projets futurs

Le projet ESPRIT □ L'expérience que Neurones a acquise dans la 2D (dessins animés) et dans la 3D (séries d'animation 3D et cyberacteurs) a fait naître un projet qui fait intervenir les deux dans

une nouvelle approche.

Le marché de la série 2D est beaucoup plus prometteur que celui de son équivalente 3D. Cependant, la technique de création de dessins animés, même informatisée, est très lourde actuellement. En effet, elle nécessite la génération de 24 dessins à la main pour former une seconde de film, sans que ceux-ci soient réutilisables dans d'autres plans ou dans d'autres épisodes. Il en va tout autrement pour créer une série en images de synthèse. Les modèles tridimensionnels sont créés une fois pour toutes, sont rassemblés dans une base de données à partir de laquelle tous les épisodes sont conçus. Le reste du travail n'en reste pas moins complexe — mise en scène, animation des personnages, tournage avec la caméra synthétique... — mais le gain de productivité reste très appréciable si on le compare au processus de fabrication d'une série 2D.

Une nouvelle technique, celle du «rendu en ligne claire», permet de produire une animation ayant un «look» dessin animé à partir d'une animation créée en 3D. L'idée de Neurones est de fabriquer des séries 2D grâce à un nouveau processus qui incorporerait des outils 3D et qui serait donc nettement plus productif. Une solution logicielle est actuellement développée à Liège pour supporter ce processus : elle fait intervenir les logiciels internes Gabby et Trag, mais aussi des logiciels commerciaux comme 3D Studio MAX et Softimage. Une configuration possible de cette solution, dont la forme finale n'est pas encore déterminée, est présentée à la figure 2.13.

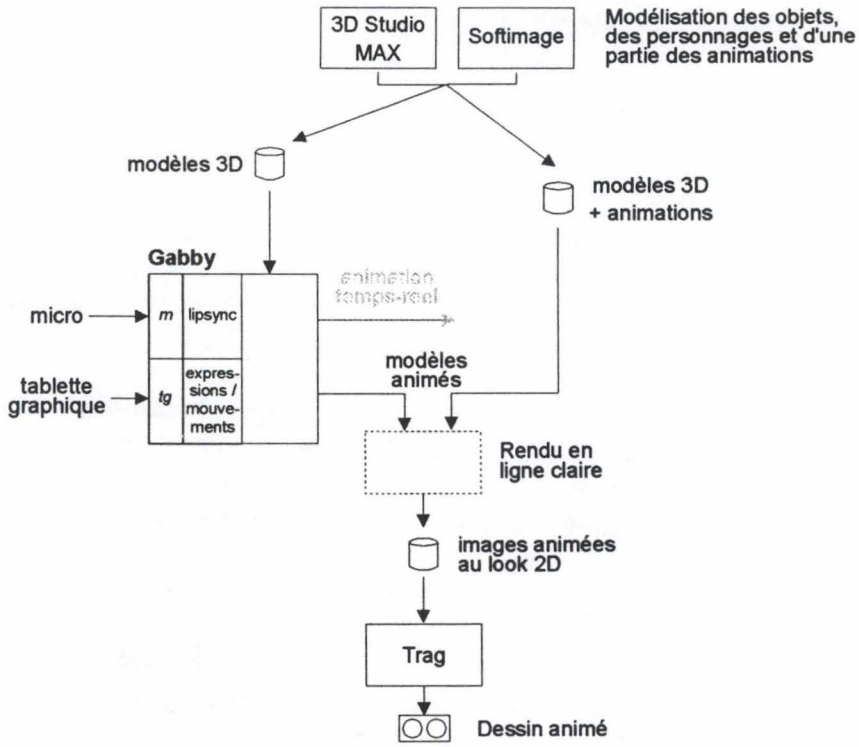


Figure 2.13 Une solution logicielle pour le projet ESPRIT

Sur cette figure, la modélisation des objets 3D est réalisée avec 3D Studio MAX et Softimage. Les animations de ces objets sont soit réalisées avec les techniques «classiques» implémentées dans ces deux mêmes logiciels, soit avec la technique originale de Gabby — le micro et la tablette. Objets et animations sont alors traités par un module qui produit leur rendu en ligne claire. Les images qui en résultent peuvent alors être retouchées et organisées par Trag pour produire le dessin animé final.

Ce projet est subventionné par la Communauté Européenne dans le cadre de son programme ESPRIT.

Figure 2.14
DocToon

DocToon □ Le projet DocToon vise à introduire les marionnettes virtuelles dans les hôpitaux et a été sélectionné par la Région Wallonne en 1996 dans le cadre de son programme «Du Numérique au Multimédia». Son objectif principal est d'améliorer les conditions de l'enfant hospitalisé par une familiarisation avec l'hopital et une dédramatisation contribuant à réduire les conséquences négatives immédiates et différées de l'hospitalisation. En outre, DocToon (figure 2.14) constituerait un média efficace pour l'éducation à la santé, et pourrait améliorer les conditions de travail du personnel médical et infirmier.

Ce projet se développe en collaboration avec les Facultés de Psychologie et de Médecine de l'ULg, et avec le CHR de la Citadelle à Liège. Un premier DocToon est déjà installé dans le CHR depuis quelques mois ; les échos de l'expérience sont rapportés dans [Col98].

Le studio Transmédia □ L'idée du studio transmédia est de fournir, à partir de la technologie des cyberacteurs, un service original de communication et de pénétration de l'information. La communication interne ou externe d'une entreprise et l'enseignement sont entre autres visés.

Techniquement parlant, un animateur, dans un studio fixe situé par exemple à Liège, serait aux commandes d'une marionnette et aurait à sa disposition graphiques, pubs, clips ou autres moyens multimédia, qui seraient autant d'ingrédients pour réaliser une sorte d'émission télévisée interactive. L'entreprise cliente du service aménagerait une salle de représentation dans le lieu de son choix — dans laquelle elle installerait juste une télévision et une caméra— et se brancherait par modem sur le studio liégeois pour y capter l'émission. Ce système est actuellement en phase prototype.

Le cyberacteur, la forme, le contenu et l'heure de diffusion de la présentation interactive seraient déterminés par l'entreprise en fonction de ses objectifs de communication, et l'animateur serait éventuellement un de ses membres. La «philosophie transmédia» se réclame d'une réelle intégration entre les différents média utilisés, le tout dans une présentation temps réel interactive.

Chapitre 3
Le projet GabbyCapture

Le chapitre 3 a pour but de présenter le projet GabbyCapture, qui s'est déroulé du 1er septembre 1997 au 31 janvier 1998. Nous commencerons par présenter les objectifs du projet (3.1), puis nous exposerons les réalisations auxquelles il a donné lieu (3.2).

3.1. Objectifs du projet

Le projet GabbyCapture répondait à deux objectifs.

Le premier concernait les cyberacteurs (et les projets qui en sont dérivés, comme DocToon et le studio Transmédia) : il s'agissait de doter ces cyberacteurs de mouvements corporels plus élaborés. En effet, comme nous le faisions remarquer dans le chapitre 2, les moyens qui permettent d'animer en temps réel un cyberacteur —un micro et une tablette graphique— sont relativement bien adaptés pour générer des expressions faciales, mais le sont beaucoup moins dès qu'il s'agit d'appliquer des déplacements et des mouvements corporels.

Le second objectif est lié au projet ESPRIT. Ce projet implique l'utilisation de techniques 3D dans la chaîne de production de séries d'animation 2D, et ce afin d'augmenter la productivité de cette chaîne. Parmi les techniques d'animation 3D, certaines sont également plus productives que d'autres. Le second objectif de GabbyCapture était de doter Neurones d'une technique d'animation susceptible d'augmenter la productivité des techniques 3D.

Pour pouvoir atteindre ces deux objectifs, Neurones a acheté un *système de capteurs*—le système MotionStar, de la firme Ascension Technologies, inc. Il s'agit d'un dispositif qui permet d'enregistrer des données de base concernant les mouvements de la personne ou de l'objet sur lequel il peut être fixé. Pour être plus précis, ce système comporte un certain nombre de capteurs, que l'on fixe généralement sur une personne, à proximité des articulations de son corps : cou, poignets, coudes, épaules, etc. Le système est capable, sous certaines conditions, de mesurer la position et l'orientation

des capteurs et de communiquer ces mesures à un ordinateur. Nous reviendrons largement sur son fonctionnement par la suite.

Ce système de capteurs a donc été à la base de la solution technique qui a permis (nous verrons dans quelle mesure) d'atteindre les objectifs du projet. En ce qui concerne les cyberacteurs, l'idée de cette solution était d'enregistrer les mouvements d'un être humain et de les reproduire en temps réel sur un cyberacteur, tout en animant son visage avec les micro et tablette habituels. En ce qui concerne le projet ESPRIT, l'idée était de créer, sur base des mouvements effectués par un être humain et enregistrés par le système de capteurs, les animations des personnages 3D qui servent de base pour réaliser les séries d'animation 2D. Cette technique est plus performante que celles utilisées habituellement; de plus, elle permet d'obtenir des mouvements plus naturels.

Gabby réalise l'animation temps-réel des cyberacteurs ; d'autre part, il participe à la solution logicielle développée dans le cadre du projet ESPRIT. Une première partie du projet GabbyCapture a donc consisté à réaliser une extension à Gabby —appelée également GabbyCapture—, pour que Gabby soit à même de traiter les données en provenance du système de capteurs, de telle sorte qu'il puisse satisfaire aux deux objectifs du projet GabbyCapture. Celui-ci tire d'ailleurs son nom de «Gabby» et de «capture de mouvements». La figure 3.1 présente un Gabby auquel GabbyCapture a été ajouté. Sur cette figure, on voit que GabbyCapture est constitué de deux parties : d'une part, le pilote 'sM' qui récupère les données du système MotionStar, et, d'autre part, le module 'mouvements' qui utilise les données de 'sM' pour modifer la posture du cyberacteur. L'output «temps réel» de Gabby est utilisé dans le cadre de l'«objectif cyberacteurs», tandis que l'output généré à des fins d'enregistrement d'animations serait utilisé dans le cadre de l'«objectif projet ESPRIT» (voir la figure 2.13).

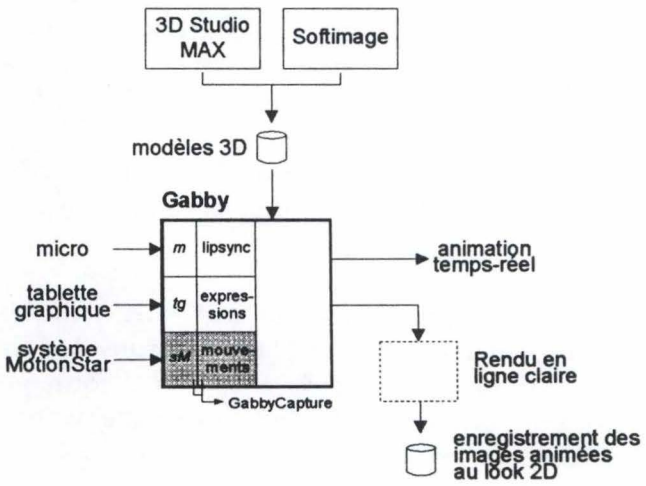


Figure 3.1 Gabby avec GabbyCapture

L'existence éventuelle de pilotes de périphériques pour MotionStar, disposant d'une interface de programmation, pourrait nous économiser l'écriture de sM. Une partie du projet GabbyCapture a donc consisté à rechercher de tels pilotes.

3D Studio MAX et Softimage sont utilisés dans le projet ESPRIT pour modéliser les objets et personnages 3D et une partie de leurs animations. Une autre partie du projet GabbyCapture a donc consisté à rechercher d'éventuels pilotes ou modules permettant d'interfacer le système MotionStar avec ces deux logiciels, pour pouvoir réaliser avec eux des animations sur base de mouvements d'êtres humains.

La hiérarchie des objectifs présentée à la figure 3.2 reprend tous les objectifs [1] du projet.

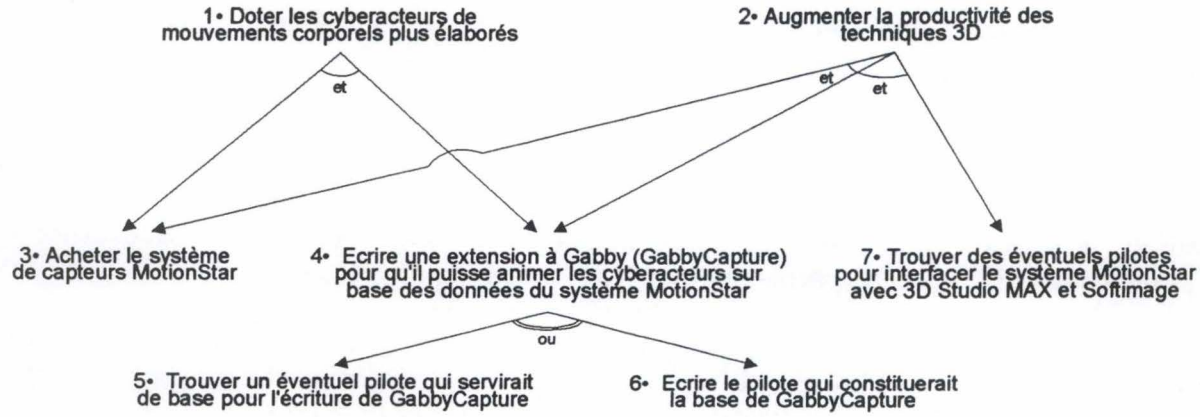


Figure 3.2 Hiérarchie des objectifs du projet

3.2. Réalisation du projet

Le système MotionStar étant déjà acheté, il restait à remplir les objectifs 4 à 7 de la figure 3.2. Deux personnes ont été chargés de le faire, à partir du 1er septembre 1997 : Jean-Paul Schäfer et moi-même, sous la responsabilité de Sébastien Dohogne. Puisqu'il concernait Gabby, ce projet faisait partie de l'activité de recherche et de développement relative aux cyberacteurs et était donc réalisé dans le département cyberacteurs de Neurones Animation SA, à Liège.

La hiérarchie des objectifs présentée à la figure 3.2 n'a été constituée qu'après la réalisation du projet. Au début de celui-ci, la situation n'était pas aussi claire et certaines données ne pouvaient pas être connues à l'avance.

Le projet a débuté par l'écriture, par Jean-Paul Schäfer et moi, de GabbyCapture (les objectifs 4 et 6 de la figure 3.2). Deux mois et demi plus tard (aux environs du 15 novembre 1997), cette écriture a été interrompue et la recherche des logiciels existants a commencé —elle correspondait à la réalisation des objectifs 5 et 7 de la figure 3.2. Elle a été effectuée par moi seul, Jean-paul Schäfer étant assigné à un projet prioritaire, et s'est terminée au 31 janvier 1998, date de la clôture du projet.

Le projet GabbyCapture a produit deux résultats :

- la réalisation d'une première version de GabbyCapture ;
- la découverte d'un certain nombre de logiciels fonctionnant avec le système MotionStar.

Les deux sections qui vont suivre présentent ces deux résultats, le travail qui a été fourni pour pouvoir les atteindre, ainsi que la mesure dans laquelle ils ont rempli les objectifs du projet.

[1] Par «objectifs», on entend les objectifs ainsi que les moyens d'y parvenir. En effet, un élément de la hiérarchie des objectifs représente un objectif, qu'on remplira en réalisant son (ou ses) élément(s) «fils» dans la hiérarchie, mais représente aussi un moyen de parvenir à la réalisation de son (ou ses) élément(s) «père(s)» dans cette hiérarche.

Nous allons voir qu'une partie des objectifs reste encore à atteindre. C'est la raison pour laquelle le projet GabbyCapture a été, après le 31 janvier 1998, «incorporé» et prolongé dans le cadre du projet ESPRIT.

3.2.1. *GabbyCapture*

La version de GabbyCapture qui a été développée n'est pas intégrée à Gabby : elle fonctionne comme une application indépendante. Elle permet d'afficher à l'écran un cyberacteur et de «connecter» celui-ci au système de capteurs selon des modalités déterminées par l'utilisateur de GabbyCapture, pour que le cyberacteur réalise, en temps réel, les mêmes mouvements que ceux effectués par l'être humain sur lequel les capteurs du système sont attachés. Les fichiers à partir desquels le cyberacteur est chargé et affiché dans GabbyCapture sont bien sûr les mêmes que ceux utilisés par Gabby. La structure de données qui accueille le cyberacteur dans GabbyCapture aurait dû être la même que celle de Gabby. Nous n'avons cependant pas eu de contacts avec la personne qui a écrit Gabby, et nous avons écrit notre propre structure de données, tout en sachant que, dans une phase ultérieure, nous devrions homogénéiser les deux structures.

Un système de calibration a été prévu, afin d'adapter les données qui proviennent du système de capteurs par rapport au cyberacteur à animer. La calibration est nécessaire à cause du fait que les capteurs sont placés de manière approximative sur les articulations de l'être humain, et que celui-ci peut avoir une morphologie différente de celle du cyberacteur.

Le système de capture de mouvements entier, comprenant GabbyCapture et le système de capteurs, est présenté à la figure 3.3.

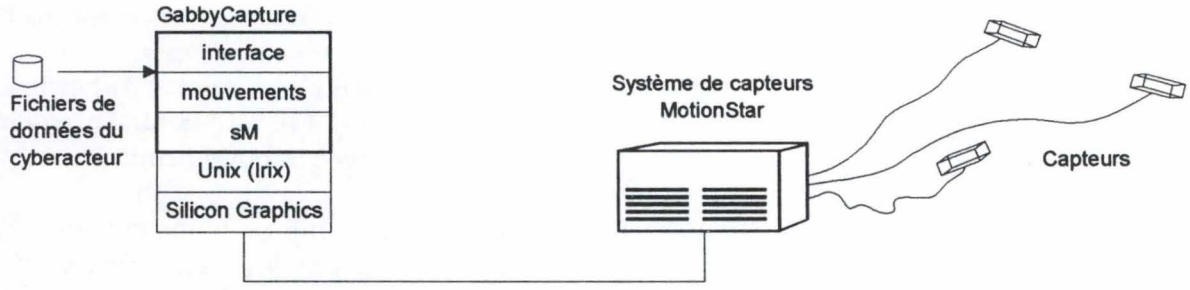


Figure 3.3 Le système de capture de mouvements

Les capteurs du système MotionStar peuvent être fixés sur les articulations les plus importantes du corps humain, mais ne peuvent pas l'être, par exemple, sur les phalanges. Quand un animateur humain les a attachés sur lui, le système MotionStar mesure, 25 fois par seconde (ce nombre est paramétrable), la position et l'orientation des capteurs et envoie ces mesures à la machine Unix. Le système MotionStar acheté par Neurones ne contenait que 3 capteurs lorsque nous avons écrit GabbyCapture. Celui-ci a toutefois été écrit pour pouvoir traiter des données en provenance d'un nombre quelconque de capteurs.

GabbyCapture s'exécute sur une station SiliconGraphics fonctionnant sous Unix (tout comme Gabby, d'ailleurs). Le module 'sM', 25 fois par seconde, récupère un message de données en provenance du système MotionStar et met ces données à disposition du module 'mouvements' qui,

sur base du cyberacteur qui est chargé à partir de fichiers Gabby et sur base du message de données de 'sM', calcule la posture que le cyberacteur doit prendre. Le module 'interface' est utilisé pour afficher ce cyberacteur ; bien entendu, cet affichage ne sera plus nécessaire lorsque GabbyCapture sera intégré à Gabby. Le module 'interface' permet aussi à l'utilisateur du système :

- de décider quel cyberacteur charger ;
- de choisir les parties du corps du cyberacteur auxquelles les mouvements de l'être humain vont être appliquées : l'entièreté du corps, une partie seulement, voire une seule articulation. Pour pouvoir calibrer de manière précise une articulation du cyberacteur, par exemple, il est préférable que les autres articulations ne bougent pas.
- L'utilisateur peut aussi choisir quels capteurs vont faire bouger les articulations du cyberacteur ; il aura demandé à l'animateur humain de fixer ces capteurs sur ses articulations qui correspondent à celles du cyberacteur.
- L'utilisateur peut lancer l'animation du cyberacteur ; pendant que l'animateur humain et le cyberacteur exécutent leurs mouvements, l'utilisateur peut «calibrer» les mouvements du cyberacteur.

Voici un scénario typique d'exécution.

1• L'utilisateur charge, par exemple, le cyberacteur Oscar dans GabbyCapture. Oscar apparaît à l'écran.

2• L'utilisateur veut uniquement animer le bras droit d'Oscar, qui est composé de la partie supérieure du bras droit, de l'avant-bras droit et de la main droite.

3• Après avoir dit à l'animateur humain de fixer les capteurs 1, 2 et 3 respectivement sur son épaule droite, sur son coude droit et sur son poignet droit, l'utilisateur, dans GabbyCapture, «associe» ces capteurs avec les 3 membres correspondants d'Oscar.

4• L'utilisateur lance uniquement l'animation de la partie supérieure du bras droit d'Oscar. MotionStar n'envoie alors que les données concernant le capteur 1. L'utilisateur demande à l'animateur humain d'effectuer quelques mouvements de base dans chaque direction, afin de calibrer les mouvements d'Oscar jusqu'à ce qu'ils soient satisfaisants.

5• L'utilisateur lance l'animation de la partie supérieure du bras droit *et* de l'avant-bras. Le système MotionStar envoie alors les données des capteurs 1 et 2. Quand le mouvement est calibré pour ces 2 membres, l'utilisateur recommence avec la main droite.

Les modules 'sM', 'mouvements' et 'interface' présentés à la figure 3.3 ne constituent qu'une première ébauche de l'architecture de GabbyCapture. Leur but n'est pas de présenter cette architecture, mais simplement de faire comprendre les fonctionnalités offertes par GabbyCapture, dans une présentation qui est similaire à celle que nous avons faite pour Gabby (modules 'm', 'lipsync', 'tg' et 'expressions' sur la figure 3.1).

GabbyCapture, tout comme Gabby, a été écrit en C et en C++, avec la bibliothèque graphique OpenGL, sur une station Silicon Graphics fonctionnant sous Unix. Son interface a été écrite en X Window. Nous donnerons une description détaillée de GabbyCapture dans les chapitres ultérieurs.

La méthode que nous avons utilisée pour développer GabbyCapture a essentiellement consisté à écrire directement le code de l'application. De temps à autre, un problème nous amenait à devoir l'analyser de manière plus mathématique ; parfois, aussi, certains aspects de l'architecture du logiciel ont dû être mis sur papier.

L'écriture de GabbyCapture m'a fait réfléchir à la question suivante : n'y aurait-il pas moyen

d'adopter une méthode de développement et un langage qui permettraient de réaliser une analyse globale, *top-down*, depuis le problème jusqu'à sa solution informatique ? Une telle approche n'apporterait-elle pas une meilleure compréhension du problème, et n'éclairerait-elle pas les choix à réaliser durant tout le processus de développement ? Les chapitres 4 et 5 tentent de répondre à cette question. En effet, dans ces chapitres, nous avons essayé d'adopter, *a posteriori*, cet autre type d'approche en réalisant une spécification mathématique du problème de la capture de mouvements (chapitre 4) et en donnant une description, à l'aide du langage ROOM [1], de GabbyCapture (chapitre 5). Le chapitre 5 confronte les deux approches et essaie de mettre en évidence les problèmes engendrés par l'écriture directe du code et les solutions qui sont apportées par l'adoption d'une méthode de développement plus rigoureuse.

De manière générale, les méthodologies de développement de logiciels et les langages de plus «haut niveau» ont d'abord été développés dans les domaines classiques de l'informatique, tels l'informatique de gestion. Des méthodes et des langages (comme ROOM) existent aussi dans le domaine du temps réel, mais sont plus récents et moins répandus. Cela explique qu'aucune méthode d'analyse, à l'heure actuelle, n'ait encore satisfait Neurones. Depuis peu, cependant, une étude est en cours dans l'entreprise liégeoise : elle pourrait aboutir à l'introduction, chez Neurones, d'une méthode de développement, d'un langage et d'un outil de développement le supportant.

Le code de GabbyCapture a été écrit en C et en C++.

Historiquement, les programmes temps-réel étaient écrits avec des langages comme C, certaines parties étant même écrites en assembleur pour des questions de vitesse d'exécution. L'évolution de la puissance des machines permet maintenant d'adopter des langages plus structurés, comme C++. Neurones a suivi cette évolution : Gabby, initialement écrit en C, a progressivement été réécrit en C++ et est actuellement entièrement refondu en C++. Cependant, si C++ fournit tous les concepts du paradigme orienté objet, il n'oblige absolument pas le programmeur à les adopter. Le programmeur habitué à écrire du code C peut donc garder ses habitudes, même s'il utilise C++. Cela explique pourquoi notre code est «hybride»: il contient des classes C++ mais ne peut pas être qualifié de «pur» orienté-objet.

Si on se réfère aux modules de GabbyCapture tels qu'ils sont identifiés sur la figure 3.3, on peut dire que Jean-Paul Schäfer a presque entièrement réalisé le module 'interface', que j'ai presque entièrement réalisé le module 'sM' et que nous avons participé tous les deux aux développement du module 'mouvements'.

Comment situer GabbyCapture par rapport aux objectifs du projet ? La version de GabbyCapture qui a été écrite remplit l'objectif 6 de la figure 3.2 et remplit partiellement l'objectif 4 de la même figure. Ces objectifs concernaient l'écriture d'une extension à Gabby pour que celui-ci puisse animer un cyberacteur sur base des données de mouvements enregistrées par le système MotionStar. Pourquoi «partiellement» ? Tout d'abord parce que GabbyCapture n'est pas intégré à Gabby ; ensuite, parce que le système de calibration mis au point dans notre version de GabbyCapture n'est pas suffisant pour pouvoir adapter les mouvements d'un être humain sur ceux d'un cyberacteur. Le problème des différences de morphologie entre humain et cyberacteur reste irrésolu : différences de taille entre les membres humains et virtuels, différences concernant les limites d'angles qui peuvent être déterminés par les articulations, etc.

[1] Le langage ROOM – Real-time Object-Oriented Modeling – sera introduit dans le chapitre 5.

3.2.2. *Logiciels existants*

La recherche de logiciels existants s'est principalement effectuée sur Internet. J'y ai trouvé un certain nombre de sites concernant la capture de mouvements. Le site de la firme Ascension Technologies ([Ascension]*), de laquelle provient le système MotionStar, était particulièrement intéressant : il contenait une liste d'entreprises ayant développé des pilotes ou logiciels fonctionnant avec leurs systèmes de capteurs.

Sur base de cette recherche et des contacts qui ont suivi, trois logiciels ont retenu mon attention.

MetaTrack Capture □ Le premier, MetaTrack Capture, de la firme américaine Win Systems, était un pilote de périphérique pour le système MotionStar, qui se présentait sous la forme de classes C++. La spécification qui m'avait été envoyée présentait MetaTrack comme étant doté de fonctionnalités similaires à celles du GabbyCapture que nous avions écrit. De plus, MetaTrack existait aussi sous la forme de pilotes pour Softimage et 3D Studio MAX : il aurait donc pu servir à la fois de base pour développer GabbyCapture et pour interfacer le système de capteurs avec les deux logiciels commerciaux utilisés par Neurones. Le prix de ces 3 pilotes défiait toute concurrence.

Malheureusement, l'évaluation de la version de démonstration que j'ai reçue par la suite a montré que les fonctionnalités de MetaTrack étaient bien plus limitées que celles de GabbyCapture. De plus, l'impossibilité de pouvoir disposer de versions d'évaluation pour 3D Studio MAX et Softimage m'a fait comprendre que les pilotes pour ces deux logiciels n'étaient encore qu'en phase de développement.

LambSoft □ Le logiciel LambSoft de la firme Lamb & Co se présentait sous la forme d'un module (un *plug-in* dans la terminologie MAX) pour 3D Studio MAX. Il paraissait doté de fonctionnalités évoluées concernant la calibration des données sur les personnages 3D, mais coûtait très cher et n'était disponible qu'à partir du premier semestre 1999.

FiLMBOX □ Le troisième et dernier logiciel fut le seul à être réellement intéressant. Il s'agissait du logiciel FiLMBOX de la firme canadienne Kaydara ([Kaydara]*). Bien plus qu'un simple pilote, ce logiciel est doté de fonctionnalités similaires à celles de Gabby. Une comparaison entre Gabby et FiLMBOX fait d'ailleurs l'objet de la figure 3.4. En ce qui concerne FiLMBOX, cette comparaison se base sur les tests que j'ai réalisés sur une version d'évaluation.

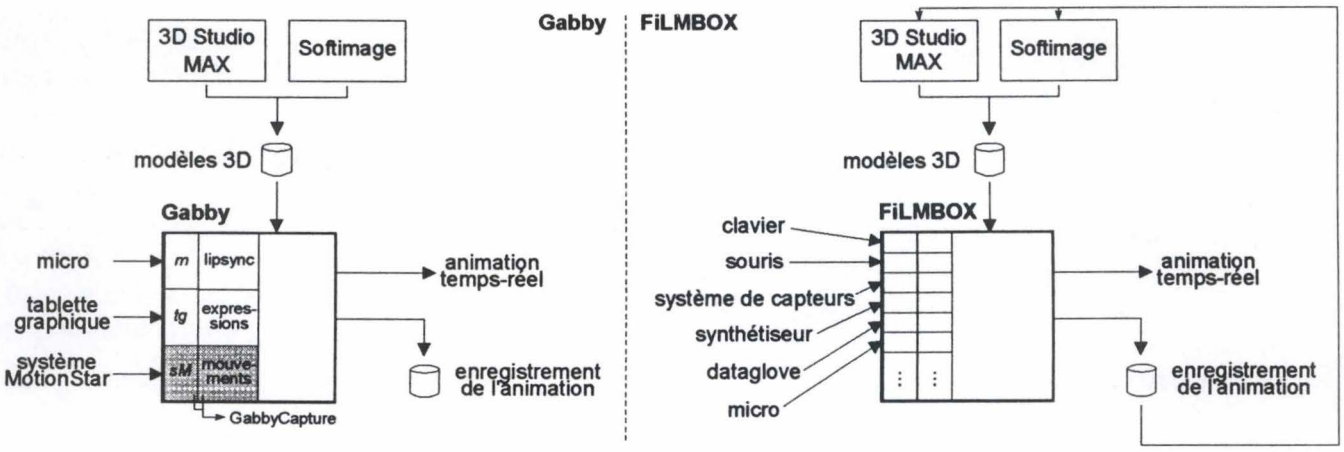


Figure 3.4 Gabby et FiLMBOX

Comme nous le disions au chapitre 2, la phase de modélisation des cyberacteurs est réalisée avec Softimage et 3D Studio MAX. Les modèles ainsi créés sont importés dans Gabby qui, sur base d'un micro et d'une tablette (et d'un système de capteurs si on tient compte de son module GabbyCapture), sont animés en temps réel. Gabby peut également enregistrer l'animation créée.

FiLMBOX est également capable d'importer des modèles de 3D Studio MAX et de Softimage, de les animer en temps réel ou d'enregistrer l'animation.

La première différence entre FiLMBOX et Gabby se situe au niveau de la facilité d'importation et de réexportation de et vers Softimage et 3D Studio MAX. La phase d'importation des modèles dans Gabby est semi-automatisée, tandis que les animations enregistrées ne sont plus réexportables vers les deux logiciels. FiLMBOX, quant à lui, importe et exporte les modèles et les animations de et vers Softimage automatiquement. Les échanges sont moins faciles mais néanmoins réalisables, dans une certaine mesure, avec 3D Studio MAX.

Une autre différence réside dans la variété des périphériques d'entrée supportés. Les possibilités de FiLMBOX à ce niveau sont importantes : il peut supporter presque tous les types de périphériques utiles pour animer un personnage —souris, clavier, systèmes de capteurs, micro, dataglove, joysticks et même synthétiseurs musicaux. Les modules qui permettent de connecter les données provenant des périphériques d'entrée à des modèles 3D (qui sont représentés sur la figure 3.4 par la deuxième colonne de cases, en partant de la gauche, dans la boîte de FiLMBOX) sont particulièrement efficaces: toutes les combinaisons peuvent être utilisées pour animer un personnage. FiLMBOX peut, par exemple, animer

- le corps d'un personnage sur base des données d'un système de capteurs dont les capteurs sont fixés sur une personne ;
- les mains du personnage sur base des données provenant de deux datagloves que la personne a enfilés ;
- les mouvements de lèvres sur base d'un micro (et d'un algorithme de synchronisation labiale similaire à celui de Gabby) ;
- les expressions du visage sur base des données en provenance d'une caméra filmant les expressions de la personne.

Pour cet exemple idéal, il faut bien sûr disposer de tous les périphériques adéquats ; on peut cependant trouver des solutions plus économiques, comme le fait Gabby avec sa tablette (que FiLMBOX ne supporte pas), ou comme peut le faire FiLMBOX avec un joystick, le clavier, la souris... On peut dire que FiLMBOX est nettement plus souple que Gabby concernant l'animation des personnages 3D. Cependant, la version de démonstration de FiLMBOX n'incluait pas encore l'algorithme de synchronisation labiale.

Les performances d'affichage des deux logiciels sont très différentes : alors que Gabby, s'exécutant sur la Silicon Graphics la plus puissante, peut afficher en temps réel des cyberacteurs constitués au plus de 4000 polygones, FiLMBOX, sur une machine de puissance similaire, peut animer en temps réel des modèles dépassant 22 000 polygones, même avec des volumes de données importants en entrée (système de capteurs, datagloves, etc).

Le prix de FiLMBOX est tout à fait concurrentiel si on le compare au coût entraîné par le développement et la maintenance de Gabby.

Le module de FiLMBOX qui concerne la manipulation des données en provenance des systèmes de capteurs supportés (dont le système MotionStar) est bien plus complet que la version de

GabbyCapture et dispose notamment de possibilités de connexion aux modèles 3D et de calibration bien plus évoluées. Ces possibilités sont tout à fait satisfaisantes pour adapter les mouvements d'un être humain à ceux d'un cyberacteur, même de morphologies différentes.

Un module de FiLMBOX, qui n'était pas encore inclus dans la version de démonstration, concernait la réalisation de rendus en «ligne claire», en temps réel ou non.

FiLMBOX dispose d'un kit de développement (SDK : *Software Development Kit*) qui permet de faire évoluer FiLMBOX en y ajoutant du code. Il est notamment possible d'écrire des modules permettant d'importer dans FiLMBOX des nouveaux formats de modèles 3D. Il est aussi possible de définir des nouveaux modules pour «connecter» les modèles 3D avec les périphériques d'entrée et de rédiger des nouveaux pilotes pour ces périphériques. Cette dernière possibilité n'a cependant pas pu être testée.

Comment situer MetaTrack, LambSoft et FiLMBOX par rapport aux objectifs du projet ? MetaTrack et LambSoft, du moins dans l'état d'avancement qui était le leur au 31 janvier 1998, ne pouvaient remplir aucun des objectifs du projet.

En ce qui concerne FiLMBOX, si on reprend les termes de la figure 3.2, il est tout à fait apte à remplir l'objectif 7 concernant l'interfaçage de MotionStar avec 3D Studio MAX et Softimage. La question qui reste à poser est la suivante : FiLMBOX pourrait-il aussi remplir l'objectif 5, c'est-à-dire servir de base pour écrire GabbyCapture ?

La réponse est non. Cela nécessiterait d'«extraire» de FiLMBOX le module concernant la capture de mouvements et l'incorporer dans Gabby. Bien qu'un kit de développement existe pour FiLMBOX, il est impossible de réaliser ce genre de remaniements. Mais cette question a, en fait, peu de sens. FiLMBOX dispose de capacités nettement plus importantes, qui, plutôt que de le mettre en concurrence avec GabbyCapture, le positionnent directement comme concurrent de Gabby.

La vraie question qui se pose alors est celle-ci : pour pouvoir atteindre les objectifs 1 et 2, outre l'achat du système de capteurs, n'existe-t-il pas l'alternative suivante : soit utiliser Gabby et écrire GabbyCapture, soit utiliser FiLMBOX ?

Si on tient compte de cette question, la hiérarchie des objectifs devient celle qui est présentée à la figure 3.5.

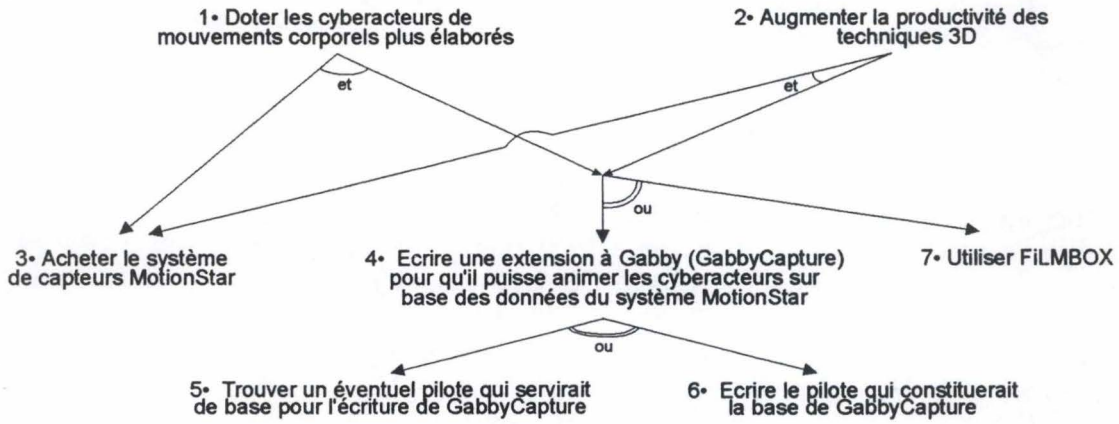


Figure 3.5 Nouvelle hiérarchie des objectifs du projet

Le nouvel objectif 7, «utiliser FiLMBOX», remplace l'ancien puisqu'il est apte à interfacer le système MotionStar avec 3D Studio MAX et Softimage. Cependant, les objectifs 4 et 7 sont maintenant *alternatifs* alors que, dans la précédente hiérarchie, ils étaient *complémentaires.*

Choisir entre FiLMBOX et Gabby, techniquement parlant, n'est pas évident du tout. Malgré les avantages que semblent présenter FiLMBOX sur Gabby, si le choix se porte sur FiLMBOX, cela signifierait qu'il faudrait réécrire un module à FiLMBOX pour que celui soit à même de réaliser tout ce que Gabby sait faire et que FiLMBOX ne fait pas : la synchronisation labiale et l'animation des cyberacteurs via la tablette graphique. Or dans la version de test dont je disposais, le SDK de FiLMBOX n'était pas encore suffisemment développé pour pouvoir faire évoluer FiLMBOX de cette manière.

Chapitre 4

Spécification de GabbyCapture

L'objectif du chapitre 4 est de donner une définition précise du problème de la capture de mouvements tel qu'il a été résolu par la version de GabbyCapture qui a été développée par Jean-Paul et moi. Cette définition a été rédigée après l'écriture du code, et s'inscrit, avec l'architecture qui fera l'objet du chapitre 5, dans la volonté de proposer une démarche de développement alternative à celle qui a été réalisée pour l'écriture de GabbyCapture.

Notre problème de capture de mouvements a été conditionné par deux éléments de sa solution, qui ont été donnés au début du projet GabbyCapture : le système MotionStar, d'une part, et les fichiers de données de Gabby, d'autre part. Ce problème aurait été différent si, par exemple, un autre type de système de capteurs avait été utilisé [1].

Dans le présent chapitre, nous nous préoccuperons le moins possible de la forme que prendra la solution informatique à ce problème. Nous savons déjà que nous devrons néanmoins aborder le système MotionStar et les fichiers de Gabby ; nous le ferons en décrivant ces éléments d'une manière plus «mathématique» qu'«informatique».

---

[1] MotionStar est un système de capteurs à technologie magnétique : il est capable de mesurer, en temps réel, la position et l'orientation de ses capteurs (dans les 3 dimensions) lorsqu'ils sont placés à l'intérieur d'un champ électro-magnétique généré par un autre composant du système. Ce champ peut cependant être perturbé par la présence de métaux, produisant des distorsions au niveau des données générées par les capteurs.

D'autres technologies existent, comme la technologie électro-mécanique (moins récente) ou la technologie optique. Les systèmes de capteurs basés sur cette dernière sont constitués de caméras qui filment l'animateur sur lequel les capteurs sont placés. A partir des images2D enregistrées par les caméras, le système détermine la position 3D des capteurs. Les systèmes optiques sont indifférents à la présence de métaux et déterminent la position des capteurs avec une précision qui est meilleure que celle qu'on obtient avec des systèmes magnétiques. Cependant, les données générées par les systèmes optiques ne sont pas temp-réel (à cause du traitement nécessaire des images 2D enregistrées par les caméras) et concernent uniquement la position des capteurs (leur orientation n'est donc pas mesurée). En outre, il arrive que des capteurs sortent du champ de vision des caméras.

Animer un cyberacteur sur base des données générées par un système magnétique est donc un problème différent de celui qui consiste à animer un cyberacteur sur base des données provenant d'un système optique.

Une vue densemble des technologies ainsi que de leurs applications peut être trouvée dans [Coc97] et [Rob97].

Notre spécification se fera en trois temps. Nous allons d'abord avoir besoin de définir un modèle de cyberacteur (4.1). Nous aurons ensuite besoin de modéliser ce qu'on entend par «mouvements» d'un être humain (4.2), avant de déterminer (4.3) comment il est possible d'appliquer les mouvements que nous avons définis en 4.2 pour les appliquer à un modèle du type de celui que nous avons défini en 4.1. Nous aurons alors répondu à l'objectif de ce chapitre ; nous présenterons néanmoins, également dans la section 4.3, certains problèmes qui restent irrésolus par la solution que constitue GabbyCapture.

4.1. Modélisation d'un cyberacteur

Plusieurs types de modèles existent ou peuvent être inventés pour décrire un cyberacteur. Le modèle que nous présentons ici a été choisi, d'abord en fonction du fait que son implémentation sera réalisée sur base des fichiers de cyberacteurs Gabby ; ensuite, en fonction de son aptitude à pouvoir représenter un personnage 3D qui doit être animé, en temps réel, sur base de mouvements effectués par un être humain (mouvements, qui, de surcroît, seront enregistrés par un système MotionStar).

Intuitivement, un cyberacteur, tout comme un être humain, peut être vu comme un «pantin articulé», c'est-à-dire un ensemble d'objets, reliés entre eux, qui peuvent être orientés différemment les uns par rapport aux autres.

Dans un premier temps, nous allons exposer la manière dont nous allons définir ces objets (4.1.1). Nous donnerons ensuite une première définition de cyberacteur en terme d'ensemble d'objets (4.1.2). Une définition finale fera l'objet du point suivant (4.1.3) : elle fera apparaître les liens qui peuvent exister entre les objets composant le cyberacteur. Nous terminerons (4.1.4) par évaluer l'aptitude de celui-ci à réaliser des mouvements approchant ceux des êtres humains.

Les développement mathématiques exposés dans cette section s'inspirent de [Edw97].

4.1.1. Définition des objets géométriques

Trois types mathématiques de base sont nécessaires pour spécifier des objets géométriques : scalaires, points et vecteurs.

Les **scalaires**, bien qu'ils ne soient pas un type géométrique, sont nécessaires comme unités de mesure. Nous utiliserons les réels. Les **points** sont les objets géométriques fondamentaux. Dans un système géométrique à trois dimensions, un point est un endroit dans l'espace. Le seul attribut que possède le point est l'endroit auquel il se trouve. Les **vecteurs** représentent toute «quantité» ayant une direction et une grandeur. Ils n'ont pas de position.

Les opérations que l'on peut effectuer sur ces trois types de base peuvent servir à déterminer des objets géométriques plus complexes. Par exemple,

$P(\alpha) = P_0 + \alpha d$,

où P_0 est un point arbitraire, d est un vecteur arbitraire et α un scalaire, définit une droite (figure 4.1). $P(\alpha)$ est un point pour n'importe quelle valeur de α.

Figure 4.1
Une droite

Il est aussi possible, à partir de ces trois types d'objets, de définir des segments de droite, des plans, des surfaces, des volumes, etc. Nous n'aborderons

pas ces définitions dans le cadre de ce travail.

Pour pouvoir manipuler ces objets géométriques, nous avons besoin d'exprimer les points et les vecteurs dans un certain référentiel. Dans un espace vectoriel à trois dimensions, on peut représenter tout vecteur de manière unique en termes de trois vecteurs linéairement indépendants :

$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

Les vecteurs v_1, v_2 et v_3 définissent un système de coordonnées, $[\alpha_1 \quad \alpha_2 \quad \alpha_3]'$ est la représentation de v dans ce système (α_1, α_2 et α_3 sont des scalaires).

Un espace vectoriel contient des vecteurs et des scalaires mais ne contient pas de points. Pour définir et manipuler ceux-ci, nous avons donc besoin de la notion d'espace affin, qui comprend des vecteurs, des scalaires et aussi des points. Dans un espace de ce type, pour obtenir la représentation unique d'un point, outre trois vecteurs linéairement indépendants, il est nécessaire de spécifier un point de référence —appelé origine :

$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3$

Comme dans un espace vectoriel, un vecteur est représenté par
$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

On peut réécrire ces deux expressions sous la forme de produits de matrices :

$$P = \begin{bmatrix}\beta_1 & \beta_2 & \beta_3 & 1\end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \text{ et } v = \begin{bmatrix}\alpha_1 & \alpha_2 & \alpha_3 & 0\end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

v_1, v_2, v_3 et P_0 définissent un *frame*, et les matrices-colonnes $[\beta_1 \quad \beta_2 \quad \beta_3]'$ et $[\alpha_1 \quad \alpha_2 \quad \alpha_3]'$ sont respectivement les représentations du point P et du vecteur v dans ce *frame*. Ces représentations sont appelées *coordonnées homogènes* de P et de v.

OpenGL, la bibliothèque graphique que nous utilisons pour l'écriture de GabbyCapture, fournit des objets de base : points, courbes et surfaces. Nous n'avons donc pas besoin de manipuler les définitions mathématiques de ces objets : nous devons juste connaître la forme des données que nous devons fournir à OpenGL pour que celui-ci puisse définir pour nous les points, courbes et surfaces voulus. Par ailleurs, puisque les objets qui font partie des cyberacteurs de Gabby sont, dans les fichiers sur lesquels GabbyCapture doit se baser, uniquement définis par des surfaces (des triangles et des quadrilatères), nous n'avons besoin que de connaître les données qui sont nécessaires à OpenGL pour créer des surfaces.

Pour ce faire, OpenGL a besoin des coordonnées homogènes des sommets déterminant les surfaces. Nous verrons par la suite dans quel(s) *frame*(s) nous exprimerons ces coordonnées. Par exemple, les lignes de code qui suivent définissent dans OpenGL le cube représenté sur la figure 4.2, dans le *frame* (x, y, z, P_0) de la même figure.

```
glBegin(GL_POLYGON);          // Face 1234
        glVertex3f(-1.0, 0.0, 1.0);
        glVertex3f(1.0, 0.0, 1.0);
        glVertex3f(1.0, 2.0, 1.0);
        glVertex3f(-1.0, 2.0, 1.0);
glEnd();
```

```
glBegin(GL_POLYGON);          // Face 2673
      glVertex3f(1.0, 0.0, 1.0);
      glVertex3f(1.0, 0.0, -1.0);
      glVertex3f(1.0, 2.0, -1.0);
      glVertex3f(1.0, 2.0, 1.0);
glEnd();
glBegin(GL_POLYGON);          // Face 4378
      glVertex3f(-1.0, 2.0, 1.0);
      glVertex3f(1.0, 2.0, 1.0);
      glVertex3f(1.0, 2.0, -1.0);
      glVertex3f(-1.0, 2.0, -1.0);
glEnd();
. . .  // Les 3 autres faces se définissent
       // de manière similaire
```

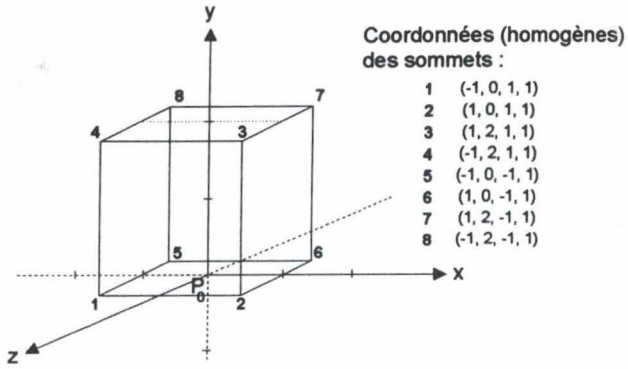


Figure 4.2 Un cube

Puisque, pour un point, la valeur de la quatrième coordonnée est toujours 1, il est inutile de la spécifier à OpenGL. La fonction glVertex3f est donc toujours appelée avec 3 arguments.

Dans le cadre de cette spécification, nous définissons donc un objet géométrique comme un ensemble de sommets, exprimés en coordonnées homogènes dans un certain *frame* ; ces sommets déterminent une (des) surface(s), qui définissent l'objet lui-même.

4.1.2. Une première modélisation d'un cyberacteur

Dans cette section, après avoir défini ce qu'on entend par «symboles» et «instances» d'objets géométriques (4.1.2.1), nous présenterons la notion de «transformation» (4.1.2.2) et verrons à quoi ces définitions peuvent nous servir pour définir un premier modèle de cyberacteur (4.1.2.3).

4.1.2.1. Symboles et instances d'objets géométriques

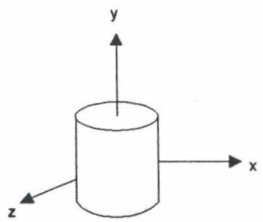


Figure 4.3
Un objet-symbole

Une première approche à la modélisation de notre personnage est de considérer celui-ci comme un ensemble d'**instances** d'objets de base qui sont vus comme des **symboles**, c'est-à-dire des objets géométriques qui sont généralement représentés à une taille et une orientation standard. Par exemple, un cylindre peut être dessiné (comme sur la figure 4.3) avec une longueur et un rayon unitaires, orienté parallèlement à un des axes, son centre correspondant à l'origine du frame. Ces objets-symboles constituent une collection de base, et pour construire notre personnage, nous allons «placer» des instances de ces symboles aux endroits voulus. Plus précisément, nous allons appliquer des **transformations** sur le symbole, pour déterminer la position, l'orientation et la taille de chacune de ses instances. Ces transformations sont illustrées sur la figure 4.4.

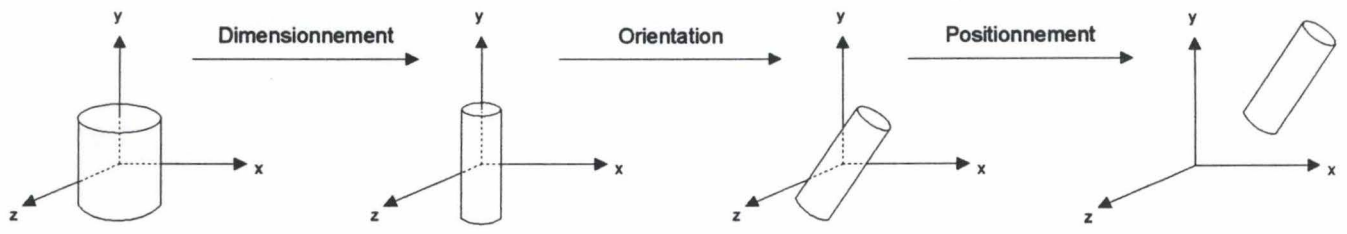


Figure 4.4 Transformations d'un symbole : création d'une instance

4.1.2.2. Transformations

Une transformation est une fonction qui, sur base d'un point (ou d'un vecteur), détermine un autre point (ou vecteur) : $Q = T(P)$ pour les points et $v = R(u)$ pour les vecteurs. Si on utilise les coordonnées homogènes, qui représentent les points et les vecteurs comme des matrices-colonnes à 4 éléments, on peut définir la transformation par une seule fontion : $\mathbf{q} = f(\mathbf{p})$, et $\mathbf{v} = f(\mathbf{u})$, $\mathbf{q}$ et $\mathbf{p}$ ($\mathbf{v}$ et $\mathbf{u}$) étant les représentations en coordonnées homogènes de points (de vecteurs).

Nous n'allons considérer que les transformations linéaires, c'est-à-dire celles qui, pour tout scalaire α et β, et pour tout point (ou vecteur) p et q, satisfont
$f(\alpha p + \beta q) = \alpha f(p) + \beta f(q)$

Cela signifie que, si on connaît les transformations linéaires de 2 sommets, on peut obtenir les transformations des combinaisons linéaires de ces sommets par la combinaison linéaire des transformations des 2 sommets. On doit calculer f uniquement pour ces 2 sommets.

Par exemple, pour une droite $P(\alpha) = P_0 + \alpha d$,
qu'on peut écrire de la forme $\mathbf{p}(\alpha) = \mathbf{p}_0 + \alpha \mathbf{d}$

($\mathbf{p}_0$ et $\mathbf{d}$ étant les représentations en coordonnées homogènes de P_0 et d), pour toute transformation linéaire f on peut écrire $f(\mathbf{p}(\alpha)) = f(\mathbf{p}_0) + \alpha f(\mathbf{d})$.

Nous n'avons besoin que des transformations de $\mathbf{p}_0$ et $\mathbf{d}$ pour pouvoir calculer la droite entière transformée.

De manière générale, pour transformer un polygone quelconque à l'aide d'une transformation linéaire, cette transformation doit uniquement être appliquée sur les sommets qui définissent ce polygone, exprimés en coordonnées homogènes.

On peut voir une transformation linéaire de deux manières : soit comme une transformation des sommets d'un objet au sein d'un même *frame*, soit comme un changement dans la représentation des sommets de l'objet, qui produit une nouvelle représentation dans un *frame* différent.

Les trois transformations dont nous avons besoin pour «placer» nos instances d'objets géométriques et définir ainsi un personnage sont des transformations linéaires. Il s'agit de la translation, de la rotation et du *scaling*. Toute transformation linéaire peut être construite à partir de ces trois transformations de base.

Translation □ Une translation est une opération qui déplace des points d'une distance fixe dans une direction donnée. Les points transformés sont donnés par $P' = P + d$ pour tout point P d'un objet translaté, d étant un vecteur. La translation est illustrée à la figure 4.5.

Dans un *frame* donné, si

$\mathbf{p} = [x\ y\ z\ 1]'$ est la représentation de P,
$\mathbf{d} = [\alpha_x\ \alpha_y\ \alpha_z\ 0]'$ est la représentation de d,
$\mathbf{p}' = [x'\ y'\ z'\ 1]'$ est la représentation de P',

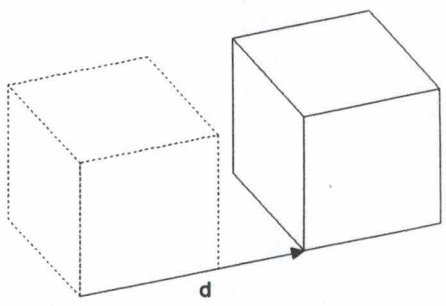


Figure 4.5 Translation d'un cube

$$\text{alors} \quad \begin{aligned} x' &= x + \alpha_x \\ y' &= y + \alpha_y \\ z' &= z + \alpha_z \end{aligned}$$, ce qui peut s'écrire **p'** = **Tp**, avec **T** = $\begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

T est une **matrice de translation**. Elle s'écrit aussi $\mathbf{T}(\alpha_x, \alpha_y, \alpha_z)$. $\mathbf{T}^{-1}$ peut être obtenue en effectuant une translation de $-d$: $\mathbf{T}^{-1}(\alpha_x, \alpha_y, \alpha_z) = \mathbf{T}(-\alpha_x, -\alpha_y, -\alpha_z)$.

Rotation □ Une rotation est une transformation plus difficile à spécifier. Commençons par la rotation d'un point par rapport à l'origine dans un plan, comme sur la figure 4.6. Sur cette figure, le point (x, y) subit une rotation d'angle θ. Les représentations polaires des points (x, y) et (x', y') dans le frame de cette figure, sont :

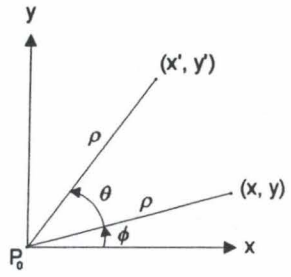


Figure 4.6 Rotation d'un point par rapport à l'origine

$$\begin{aligned} x &= \rho\cos\phi \\ y &= \rho\sin\phi \\ x' &= \rho\cos(\theta+\phi) \\ y' &= \rho\sin(\theta+\phi) \end{aligned}$$

En utilisant les formules du sinus et du cosinus de la somme de deux angles, pour les deux dernières expressions, on obtient

$$\begin{aligned} x' &= \rho\cos\phi\cos\theta - \rho\sin\phi\sin\theta = x\cos\theta - y\sin\theta \\ y' &= \rho\cos\phi\sin\theta + \rho\sin\phi\cos\theta = x\sin\theta + y\cos\theta \end{aligned}$$

On peut écrire ces équations sous forme matricielle :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

En toute généralité, une rotation est spécifiée par un point fixe (P_0 dans le cas de la figure 4.6), un vecteur autour duquel la rotation est exécutée (la rotation, sur la figure 4.6, est équivalente à une rotation par rapport à l'axe z, qui n'y est pas représenté —il sort de la feuille) et un angle de rotation (l'angle θ, sur la figure 4.6).

Dans un *frame* donné, si **p** = [x y z 1] ' est la représentation d'un point P,
p' = [x' y' z' 1] ' est la représentation d'un point P',

une rotation effectuée sur P, à partir de l'origine de ce *frame*, autour de l'axe z, d'un angle θ, produit P', et

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \\ z' &= z \end{aligned}$$, ce qui peut s'écrire **p'** = $\mathbf{R}_z$ **p**, avec $\mathbf{R}_z$ = $\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$\mathbf{R}_z$ est une **matrice de rotation**, autour de l'axe z. Elle s'écrit aussi $\mathbf{R}_z(\theta)$. On trouve, de manière similaire, les matrices de rotation autour des axes x et y :

$$\mathbf{R}_x = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ et } \mathbf{R}_y = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Si $\mathbf{R}$ dénote n'importe laquelle des 3, on a $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$. De plus, puisque $\cos(-\theta)=\cos(\theta)$ et $\sin(-\theta)=-\sin(\theta)$ on trouve que $\mathbf{R}^{-1}(\theta) = \mathbf{R}'(\theta)$.

Jusqu'à présent, nous réalisons des rotations uniquement autour de l'origine d'un *frame*. Comment, par exemple, faire une rotation du cube de la figure 4.7, d'un angle θ, autour de l'axe z, dont le point fixe serait le centre du cube, $\mathbf{p}_f$?

La solution est présentée à la figure 4.8. Elle consiste à appliquer au cube une *séquence de transformations* :

1• appliquer une translation de vecteur $-\mathbf{p}_f$ à tous les points du cube, à l'aide d'une matrice de translation $\mathbf{T}(-\mathbf{p}_f)$, et ce afin de ramener le cube à l'origine du *frame*;
2• appliquer une rotation à l'aide de $\mathbf{R}_z(\theta)$;
3• appliquer une translation $\mathbf{T}(\mathbf{p}_f)$ pour ramener le cube à sa position initiale.

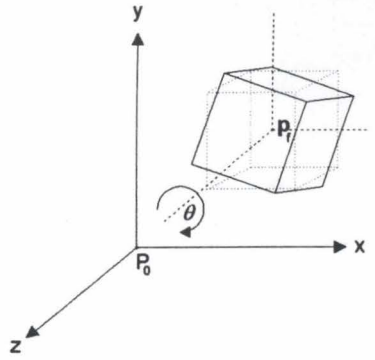


Figure 4.7 Rotation d'un cube autour de son centre

Tout point P (de représentation $\mathbf{p}$) du cube est donc transformé en un point P' (de représentation $\mathbf{p}'$) de la manière suivante :

$\mathbf{p}_1 = \mathbf{T}(-\mathbf{p}_f) \cdot \mathbf{p}$
$\mathbf{p}_2 = \mathbf{R}_z(\theta) \cdot \mathbf{p}_1$
$\mathbf{p}' = \mathbf{T}(\mathbf{p}_f) \cdot \mathbf{p}_2$

ce qui revient à écrire $\mathbf{p}' = \mathbf{M}\mathbf{p}$ avec $\mathbf{M} = \mathbf{T}(\mathbf{p}_f)\mathbf{R}_z(\theta)\mathbf{T}(-\mathbf{p}_f)$.

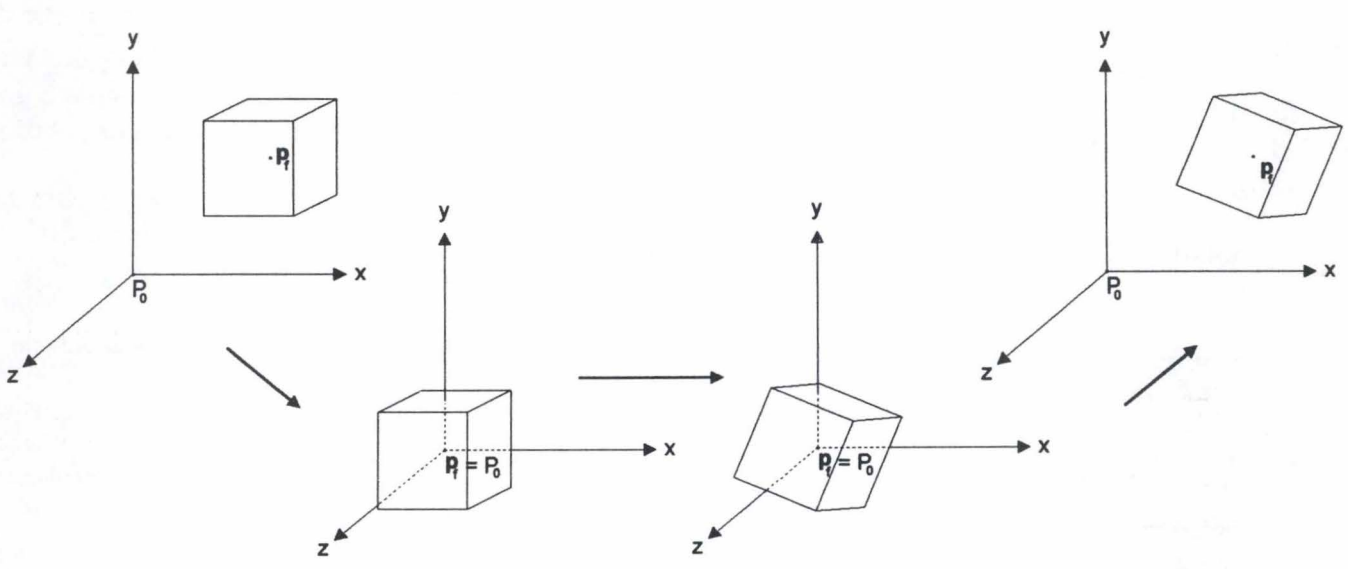


Figure 4.8 Rotation d'un cube autour de son centre : la séquence de transformations

Jusqu'à présent, nous réalisons des rotations autour d'un des trois axes d'un *frame*. Comment effectuer une rotation autour d'un vecteur arbitraire v ? On peut le faire en appliquant d'abord une rotation autour de z selon un angle α, ensuite une rotation autour de y d'un angle β, et finalement en appliquant une rotation autour de x d'un angle γ. La matrice de rotation autour du vecteur v est donnée par

$\mathbf{R}(v) = \mathbf{R}(\gamma, \beta, \alpha) = \mathbf{R}_x(\gamma)\mathbf{R}_y(\beta)\mathbf{R}_z(\alpha)$.

On trouve, dans [Edw97], aux pages 157 à 161, la manière dont les angles α, β et γ peuvent être déterminés sur base du vecteur v.

Scaling □ Le *scaling*, comme le montre la figure 4.9, est une transformation qui peut modifier les dimensions et les proportions d'un objet. Pour spécifier un *scaling*, on doit spécifier un point fixe, une direction et un facteur de *scaling* (α). Pour $\alpha > 1$, l'objet grandit dans la direction indiquée. Pour $0 \leq \alpha \leq 1$, l'objet devient plus petit dans cette direction. Les valeurs négatives d'α, comme le montre la figure 4.10, produisent une réflection par rapport au point fixe, dans la direction du *scaling*.

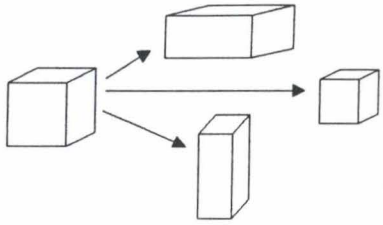


Figure 4.9 Scalings d'un cube

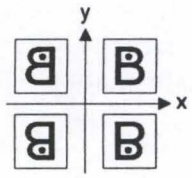


Figure 4.10
Réflexion

Dans un *frame* donné, si $\mathbf{p} = [x\ y\ z\ 1]'$ est la représentation d'un point P,
$\mathbf{p}' = [x'\ y'\ z'\ 1]'$ est la représentation d'un point P',

un *scaling* effectué sur P, à partir de l'origine de ce *frame*, dont la direction est indiquée indépendamment pour chaque axe, produit un point P', et

$$\begin{aligned} x' &= \beta_x x \\ y' &= \beta_y y \\ z' &= \beta_z z \end{aligned}$$, ce qui peut s'écrire $\mathbf{p}' = \mathbf{S}\mathbf{p}$, avec $\mathbf{S} = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

S est une **matrice de *scaling***. Elle s'écrit aussi $\mathbf{S}(\beta_x, \beta_y, \beta_z)$. $\mathbf{S}^{-1}$ est obtenue en appliquant les facteurs inverses : $\mathbf{S}^{-1}(\beta_x, \beta_y, \beta_z) = \mathbf{S}(1/\beta_x, 1/\beta_y, 1/\beta_z)$. Pour réaliser un scaling par rapport à un point arbitraire $\mathbf{p}_f$, on applique d'abord une translation d'un vecteur $-\mathbf{p}_f$, on applique un *scaling* à partir de l'origine, puis on applique une translation d'un vecteur $\mathbf{p}_f$.

4.1.2.3. Une première modélisation d'un cyberacteur

Comme nous le disions en début de section, dans une première approche, on peut considérer un cyberacteur comme un ensemble d'instances d'objets-symboles : tête, buste, avant-bras, main...

Si nous voulons définir le cyberacteur, il s'agit donc, comme le montre la figure 4.4, de dimensionner, orienter et positionner de manière adéquate les instances d'objets qui le composent. Par exemple, à partir d'un objet-symbole «main», on peut créer les deux instances qui feront partie du cyberacteur, qui auront des positions, orientations et scaling (réflexion) différents. De manière formelle, pour chaque instance, il faut donc appliquer à tout point P du symbole (de représentation **p**) la séquence de transformations qui suit pour produire un point P' de l'instance du symbole (de représentation **p'**) :

$\mathbf{p}_1 = \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{p}$
$\mathbf{p}_2 = \mathbf{R}(\theta_x, \theta_y, \theta_z) \cdot \mathbf{p}_1$
$\mathbf{p}' = \mathbf{T}(d_x, d_y, d_z) \cdot \mathbf{p}_2$,

ce qui revient à écrire **p'** = **Mp** avec $\mathbf{M} = \mathbf{T}(d_x, d_y, d_z)\mathbf{R}(\theta_x, \theta_y, \theta_z)\mathbf{S}(s_x, s_y, s_z)$.

Notre cyberacteur, jusqu'à présent, peut donc être représenté comme le tableau de la figure 4.11.

objet-symbole	translation	rotation	scaling
main	$\mathbf{T}_1(d_x^1, d_y^1, d_z^1)$	$\mathbf{R}_1(\theta_x^1, \theta_y^1, \theta_z^1)$	$\mathbf{S}_1(s_x^1, s_y^1, s_z^1)$
main	$\mathbf{T}_2(d_x^2, d_y^2, d_z^2)$	$\mathbf{R}_2(\theta_x^2, \theta_y^2, \theta_z^2)$	$\mathbf{S}_2(s_x^2, s_y^2, s_z^2)$
tête	...	...	...
buste			
avant-bras			
avant-bras			
...			

Figure 4.11 Un premier modèle de cyberacteur

Une ligne du tableau représente une instance d'objet. La première colonne contient les références vers les objets-symboles (qui sont chacun définis par un ensemble de sommets, mais qui n'apparaissent pas sur la figure) et les trois colonnes restantes contiennent les matrices de transformation —en translation, en rotation et en *scaling*.

4.1.3. Modèles hiérarchiques d'un cyberacteur

Un cyberacteur représente —au moins de manière caricaturale— un être humain et a vocation à être animé sur base des mouvements de celui-ci. Or, bien sûr, il existe des «relations» entre les différentes parties du corps d'un être humain. Par exemple, les distances entre les membres adjacents ne varient (presque) pas : ce sont les angles formés par les articulations qui varient. Tous les membres sont liés entre eux : le déplacement des jambes entraîne le reste du corps, la rotation du buste est répercutée sur les bras...[1] Il serait utile d'exploiter ce type de relations parmi les instances d'objets qui composent un cyberacteur.

Pour ce faire, la notion de graphe peut nous aider, et, en particulier, celle d'**arbre** et celle de **graphe acyclique directionnel** (GAD).

Un arbre est un graphe orienté dans lequel il n'existe aucun circuit. Tous les noeuds, sauf un — le noeud racine— sont l'extrémité terminale d'un seul arc. La figure 4.12 montre un personnage très simple et l'arbre qui le représente. Nous verrons plus loin quel type d'information on va associer aux noeuds de l'arbre. La modélisation sous forme d'arbre n'exploite cependant pas le fait que les bras gauche et droit, par exemple, sont 2 instances d'objets qui proviennent d'un même symbole et ne se distinguent que par leur position différente (ils pourraient également se distinguer par leur orientation, et aussi leur *scaling* dans le cas où, comme pour l'être humain, les membres sont identiques à une réflexion près). La notion de GAD peut alors être utile.

[1] Il est difficile de déterminer quelle partie du corps entraîne quelle autre ...

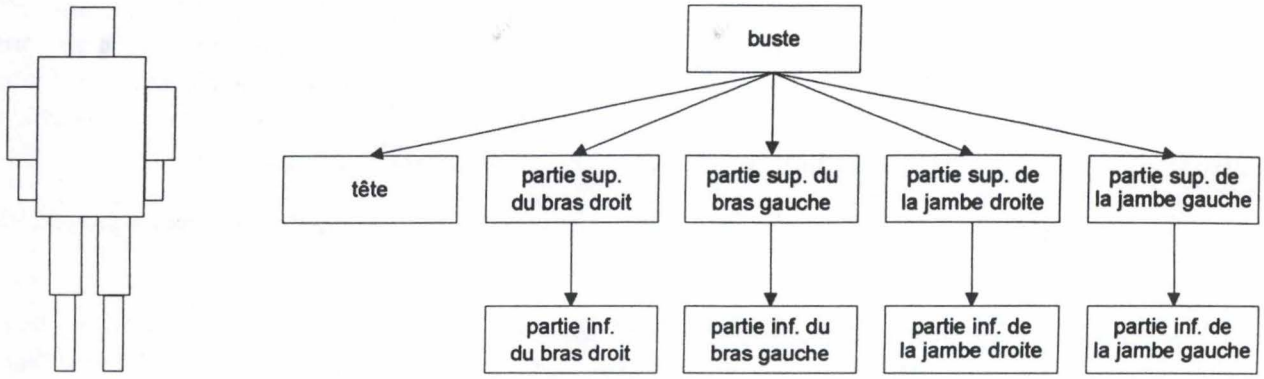


Figure 4.12 Représentation arborescente d'un personnage

Un graphe acyclique directionnel (GAD) est, tout comme un arbre, un graphe orienté dans lequel il n'existe aucun circuit. Cependant, les noeuds (sauf la racine) peuvent être l'extrémité terminale de plusieurs arcs. La figure 4.13 montre la représentation du personnage de la figure 4.12 sous la forme d'un GAD.

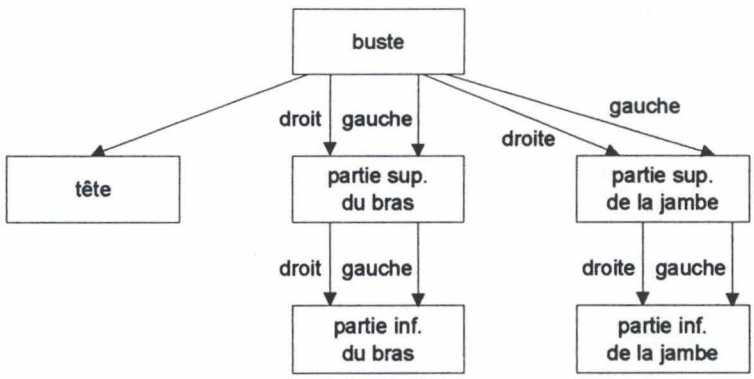
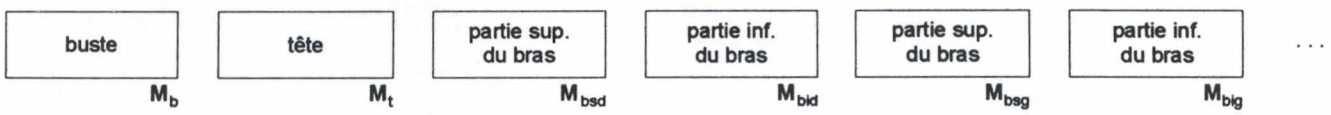


Figure 4.13 Représentation en GAD d'un personnage

Les représentations en arbre ou en GAD sont appelées **modèles hiérarchiques** d'un personnage.

Quelles informations doit-on associer à chaque noeud de l'arbre représentant un cyberacteur ? Repartons de la figure 4.12 et considérons également la figure 4.14 qui représente le personnage de la figure 4.12 avec un type de modélisation équivalent à celui de la figure 4.11, au point 4.1.2.2 (notre premier modèle de cyberacteur).



avec $\mathbf{M}_n = \mathbf{T}_n(d_x^n, d_y^n, d_z^n)\mathbf{R}_n(\theta_x^n, \theta_y^n, \theta_z^n)\mathbf{S}_n(s_x^n, s_y^n, s_z^n)$ pour n = b, t, bsd, bid, bsg, big...

Figure 4.14 Type de représentation équivalent à celui de la figure 4.11

La première information à associer à chaque noeud de l'arbre de la figure 4.12 est, bien sûr, une référence vers l'objet-symbole à partir duquel l'instance associée au noeud est définie. Si on considère que le nom écrit dans chaque noeud sur cette figure représente une telle référence (et que, par exemple, «partie sup. du bras droit» et «partie sup du bras gauche» sont deux références vers le même objet-symbole), alors l'arbre de la figure 4.12 est en fait un GAD.

Si l'on se réfère à la figure 4.14, la deuxième information à ajouter à chaque noeud devrait prendre la forme d'une matrice. Mais de quelle matrice s'agit-il ?

Si l'on veut s'inspirer du comportement du corps d'un être humain, quand le buste du cyberacteur se déplace, par exemple, la translation qu'il subit doit également être répercutée sur les autres objets du personnage et notamment sur sa tête [1]. Une idée est de définir la position de la tête, non pas par rapport au cyberacteur global, ce qui est le cas sur la figure 4.14, mais bien par rapport au buste. Pour déterminer la position correcte de la tête, il faut alors se baser sur la position du buste *et* sur celle de la tête relativement au buste :

$$\mathbf{T}_{\text{tête}} = \mathbf{T}_{\text{buste}}\mathbf{T}_{\text{tête/buste}}$$

Toute modification de la position du buste sera donc automatiquement répercutée sur celle de la tête. On peut appliquer la même idée pour déterminer l'orientation et même la dimension de la tête:

$$\mathbf{R}_{\text{tête}} = \mathbf{R}_{\text{buste}}\mathbf{R}_{\text{tête/buste}}$$
$$\mathbf{S}_{\text{tête}} = \mathbf{S}_{\text{buste}}\mathbf{S}_{\text{tête/buste}}$$

et donc $\mathbf{M}_{\text{tête}} = (\mathbf{TRS})_{\text{tête}} = (\mathbf{TRS})_{\text{buste}}\mathbf{M}_{\text{tête/buste}}$, avec $\mathbf{M}_{\text{tête/buste}} = \mathbf{M'}_{\text{tête}} = \mathbf{T}_{\text{tête/buste}}\mathbf{R}_{\text{tête/buste}}\mathbf{S}_{\text{tête/buste}}$

Alors que $\mathbf{M}_{\text{tête}}$ ($\mathbf{M}_t$ sur la figure 4.14) constituait la matrice de transformation **globale** de la tête —c'est-à-dire la matrice permettant de positionner, d'orienter et de dimensionner la tête par rapport au cyberacteur— $\mathbf{M'}_{\text{tête}}$ constitue la matrice de transformation **locale** à la tête, permettant de positionner, d'orienter et de dimensionner celle-ci par rapport au buste.

Si on applique l'idée récursivement, pour un noeud quelconque *n* du GAD représentant un cyberacteur, $\mathbf{M'}_n$ détermine la matrice de transformation locale du noeud permettant de positionner, orienter et dimensionner l'objet associé au noeud par rapport à son noeud père. Le GAD de la figure 4.12, selon cette idée, devient le GAD représenté à la figure 4.15.

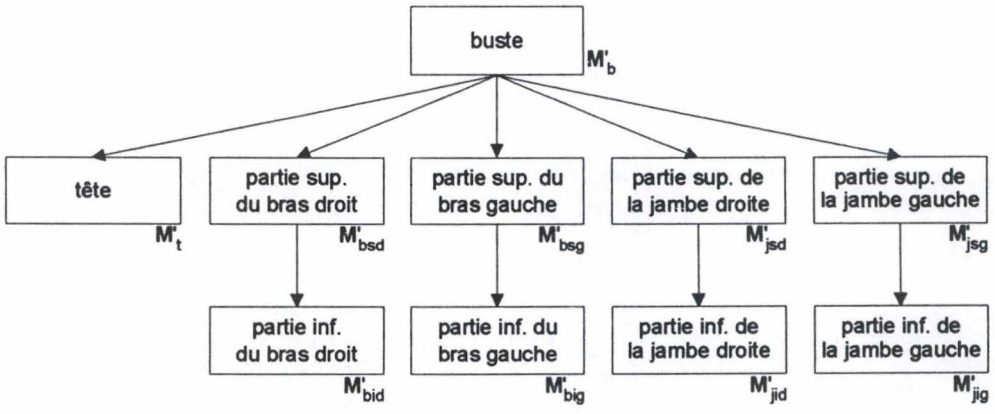


Figure 4.15 Le nouveau GAD

[1] Décider que le mouvement du buste entraîne celui de la tête est un choix arbitraire. On aurait pu faire le choix inverse. Que ce soit dans un sens ou dans l'autre, l'important est de définir la relation que la tête et le buste entretiennent.

Sur cette figure, la matrice de transformation $\mathbf{M'}_b$ du buste est égale à la matrice $\mathbf{M}_b$ de la figure 4.14, puisque le noeud du buste n'a pas de père (il est la racine de l'arbre). Comme nous l'avons déjà vu, la matrice de transformation globale de la tête ($\mathbf{M}_t$ de la figure 4.14) est obtenue en effectuant le produit $\mathbf{M'}_b\mathbf{M'}_t$. La matrice de transformation globale de la partie inférieure du bras droit ($\mathbf{M}_{bid}$ sur la figure 4.14) est égale à $\mathbf{M'}_b\mathbf{M'}_{bsd}\mathbf{M'}_{bid}$.

Comme nous le disions plus haut, on peut voir une transformation linéaire de deux manières : soit comme une transformation des sommets d'un objet au sein d'un même *frame*, soit comme un changement dans la représentation des sommets de l'objet, qui produit une nouvelle représentation dans un *frame* différent. Par exemple, sur la figure 4.15, $\mathbf{M'}_b\mathbf{M'}_{bsd}\mathbf{M'}_{bid}$ peut être vue comme une transformation, au sein du *frame* global du personnage, qui positionne, oriente et dimensionne la partie inférieure du bras droit dans ce *frame*, mais peut aussi être vue comme la matrice qui, à partir de la représentation de la partie inférieure du bras droit dans un *frame* local à cet objet, permet produire une nouvelle représentation de ce même objet dans le *frame* global du personnage. Cette matrice détermine alors un changement de représentation entre deux *frames*.

Le GAD de la figure 4.15 constitue un exemple du modèle final de cyberacteur que nous utiliserons dans le cadre de cette spécification. Il correspond à la représentation qu'on retrouve sur les fichiers de Gabby.

4.1.4. Animation d'un modèle hiérarchique

Une dernière question reste cependant à poser concernant ce modèle : est-il apte à pouvoir «accomplir des mouvements» d'une manière similaire à un être humain ?

Les matrices qui sont présentes sur chaque noeud d'un modèle hiérarchique déterminent la position, l'orientation et les dimensions d'une instance d'objet. Ensemble, elles définissent donc une posture du cyberacteur entier.

«Animer» un cyberacteur, ou bien lui faire faire des «mouvements», consiste à l'afficher un certain nombre de fois par seconde, à chaque fois dans une posture différente :

- ce nombre de fois par seconde doit dépasser 22 pour que l'oeil humain perçoive cet enchaînement comme une image animée de manière fluide (bien que 15 images par seconde, par exemple, produise un résultat déjà correct) ;
- modifier la posture du cyberacteur consiste à modifier le contenu de ses matrices.

La possibilité d'afficher le modèle 22 fois par seconde dépend de la taille de l'implémentation du modèle et des caractéristiques du *hardware* et du *software* sur lequel il va tourner. Ces considérations n'interviennent pas ici : notre seul souci pour l'instant est de savoir si le modèle mathématique que nous avons élaboré est apte à pouvoir représenter, théoriquement, des mouvements similaires à ceux d'un être humain. La question que nous avions posée en début de section se réduit donc à la suivante: les modifications qu'il est bien sûr possible d'envisager sur les matrices du modèle peuvent-elles être adaptées pour appliquer des mouvements à ce modèle qui soient similaires à ceux d'un être humain?

Reprenons l'exemple de la figure 4.15 et passons en revue les composantes **T'**, **R'** et **S'** des matrices **M'** associées à chaque noeud. La composante **T'** d'un noeud ne doit certainement pas être modifiée, du moins si ce noeud n'est pas le noeud racine : sa modification correspondrait visuellement à un démembrement du personnage. La modification de la composante **S'** de n'importe quel noeud

correspondrait à une variation des proportions du personnage : elle est aussi à proscrire. Il nous reste donc les possibilités suivantes :

- modifier les matrices de translation et de rotation du noeud racine ;
- modifier les matrices de rotation des autres noeuds.

Modifier la matrice de rotation de n'importe quel noeud qui n'est pas la racine a pour effet de modifier l'orientation d'un objet par rapport à celui de son noeud père —modifier, par exemple, l'orientation de la main par rapport à l'avant-bras. Si, dans cet exemple, le point fixe de la rotation qui est appliquée à la main ne correspond pas au «joint» (à l'articulation) qui relie la main à l'avant-bras, il y aura démembrement.

Nouvelle et dernière forme de la question initiale : modifier les matrices de translation et de rotation du noeud racine, et modifier les matrices de rotation des autres noeuds pour autant que le démembrement évoqué dans le paragraphe précédent soit évitable, nous permet-il d'appliquer à notre modèle des mouvements similaires à ceux d'un être humain ?

Les distances entre les articulations du corps humain qui ne sont séparées que par un seul membre sont constantes (du moins on les suppose comme telles). Ainsi en est-il du coude et de l'épaule, du coude et du poignet (du même bras, bien sûr), du genou et de la cheville (de la même jambe), etc. Les seules distances qui peuvent varier sont celles qui existent entre les articulations qui sont séparées par au moins deux membres, comme le poignet et l'épaule. Pour déterminer la position et la posture d'un être humain dans un certain «espace», il suffit donc de déterminer :

- la position et l'orientation d'*un* membre quelconque de son corps dans cet espace,
- les angles formés par toutes les articulations dont il est constitué. Les angles formés par le coude, par exemple, déterminent l'orientation de l'avant-bras, dont le coude est le centre de rotation, par rapport au bras.

Pour déterminer les mouvements d'un être humain, il suffit de déterminer, plus de 22 fois par seconde, les deux éléments ci-dessus.

Puisque la position d'un cyberacteur dans son «espace virtuel» est déterminable par :

- les matrices de translation et de rotation du noeud racine, qui s'appliquent à tous les noeuds du personnage,
- les matrices de rotation des autres noeuds, qui permettent de déterminer l'orientation des différentes instances d'objets par rapport à leur noeud père,

il est donc possible de déterminer les mouvements d'un cyberacteur pour qu'ils soient similaires à ceux d'un être humain. Une contrainte de modélisation, que nous n'avions pas encore signalée jusqu'ici, doit cependant être respectée : quand un objet-symbole du modèle est modélisé dans son *frame* local, il faut faire correspondre le centre de rotation de l'objet-symbole avec l'origine de ce *frame* local, puisque la matrice de rotation $\mathbf{R}(\alpha_x, \alpha_y, \alpha_z)$ qui est utilisée dans notre modèle pour placer une instance de cet objet définit une rotation par rapport à l'origine de ce *frame*. De cette manière, le point fixe de la rotation qui est appliquée à cet objet correspond à l'«articulation» qui le relie à l'instance de son noeud père.

4.2. Modélisation des mouvements d'un être humain

Puisque c'est avec le système MotionStar que les mouvements des animateurs humains seront enregistrés, la modélisation de ces mouvements passe obligatoirement par celle du fonctionnement du système MotionStar.

Le système MotionStar contient un certain nombre de capteurs. Il est capable de mesurer Nm fois par seconde ($20 \leq Nm \leq 90$), la position et l'orientation de chaque capteur dans un *frame F* qui lui est propre. La structure des données que MotionStar peut mesurer Nm fois par seconde fait l'objet de la figure 4.16.

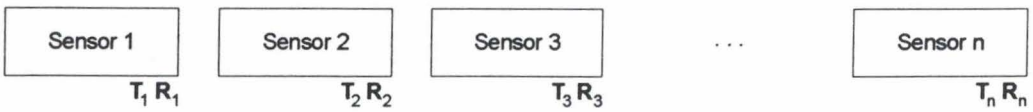


Figure 4.16 Modélisation des données provenant du système MotionStar

A chaque capteur k sont associées deux matrices :

- une matrice de translation $\mathbf{T}_k(d_x^k, d_y^k, d_z^k)$ définissant la position du capteur dans F ;
- une matrice de rotation $\mathbf{R}_k(\alpha_x^k, \alpha_y^k, \alpha_z^k)$ définissant son orientation dans le *frame F*.

Ces matrices de transformation ont la même sémantique que celles que nous avons utilisées dans le modèle d'un personnage 3D. La figure 4.16 montre cependant qu'aucune «hiérarchie» n'existe parmi les capteurs, et que toutes les matrices $\mathbf{T}_k$ et $\mathbf{R}_k$ sont globales par rapport à F.

Les capteurs du système MotionStar peuvent être fixés sur le corps d'un être humain. A quels endroits les fixer ?

On sait que, pour déterminer la position et la posture d'un être humain dans un «espace», il faut déterminer la position et l'orientation d'*un* membre quelconque de son corps dans cet espace (que l'on appelera position et orientation «globale» de l'être humain), ainsi que les angles formés par toutes les articulations dont il est constitué. On sait aussi que la position et l'orientation de la racine du modèle d'un cyberacteur est répercutée sur tous les autres noeuds.

Si on veut enregistrer la position et l'orientation «globales» de l'être humain, un capteur doit être placé sur le membre du corps de l'être humain qui correspond au noeud racine du cyberacteur, le buste dans l'exemple de la figure 4.15. La position et l'orientation globales du corps de l'être humain seront donc données par les matrices **T** et **R** de ce capteur, dans F.

Les matrices des capteurs sont des matrices globales. Les matrices **T** et **R** d'un capteur ne peuvent donc pas mesurer l'orientation d'un capteur par rapport à un second capteur, ce qui aurait été utile pour mesurer les angles formés par les articulations du corps. On aurait pu, par exemple, déterminer les angles formés par le coude en déterminant l'orientation de l'avant-bras, dont le coude est le centre de rotation, par rapport à la partie supérieure du bras.

L'information la plus pertinente que nous puissions obtenir au sujet des membres de l'être humain sont leur position et leur orientation, mesurées dans un *frame* global. Si on veut obtenir ces informations, il faut fixer un capteur sur chaque membre qui nous intéresse, le plus près possible du centre de rotation —l'articulation— du membre. La position et l'orientation du membre sur lequel le capteur est fixé sont donc exprimés dans le *frame* global ***F*** et sont données par les matrices **T** et **R** de ce capteur.

On peut donc définir ce qu'on entend par «mouvements» d'un être humain dans le cadre de cette spécification : il s'agit de séquences de positions dans le temps. Une séquence contient Nm positions par seconde, et chaque position est définie par les éléments suivants [1] :

- une matrice $\mathbf{R}_i$ et une matrice $\mathbf{T}_i$ déterminant, dans un *frame F*, la position et l'orientation globale d'un être humain,
- pour chaque partie articulée du corps de l'être humain, deux matrices, $\mathbf{T}_j$ et $\mathbf{R}_j$, $j \neq i$ pour tout j, déterminant, dans un *frame F*, la position et l'orientation de cette partie.

4.3. Appliquer les mouvements d'un être humain sur un cyberacteur

Appliquer les mouvements d'un être humain sur un personnage 3D consiste à faire adopter à celui-ci, plus de 22 fois par seconde, les «mêmes» position et posture que celles de l'être humain. Il nous suffit donc d'expliquer comment on obtient cette «équivalence» de position et de posture à un instant donné, et dire qu'il faut l'obtenir plus de 22 fois par seconde.

La figure 4.17 expose, sur un exemple, le problème et l'ensemble des données dont nous disposons jusqu'à présent pour pouvoir le résoudre.

Nous allons commencer par expliquer le principe de base de la solution (4.3.1). Nous présenterons ensuite un premier raffinement (4.3.2). A ce stade, nous aurons défini la solution telle qu'elle implémentée par GabbyCapture. Dans un dernier point, nous présenterons les problèmes qui restent encore à résoudre (4.3.3).

4.3.1. L'idée de base

L'idée est d'«associer» chaque capteur attaché sur un membre de l'être humain à l'instance d'objet du modèle 3D qui correspond à ce membre. Sur la figure 4.17, on associe donc le capteur 1 avec le buste du cyberacteur, le capteur 2 avec la partie supérieure de son bras droit, le capteur 3 avec la partie inférieure du bras droit du cyberacteur et ainsi de suite.

Cette association consiste à modifier les matrices des noeuds du modèle sur base des matrices des capteurs qui sont associés à ces noeuds.

Intuitivement, concernant le noeud racine, il suffit de remplacer ses matrices de rotation et de

[1] En fonction des besoins, il se peut que nous ne nous intéressions qu'à une partie seulement de ces éléments, et que nous appliquions au cyberacteur uniquement les mouvements du bras droit d'un être humain, par exemple.

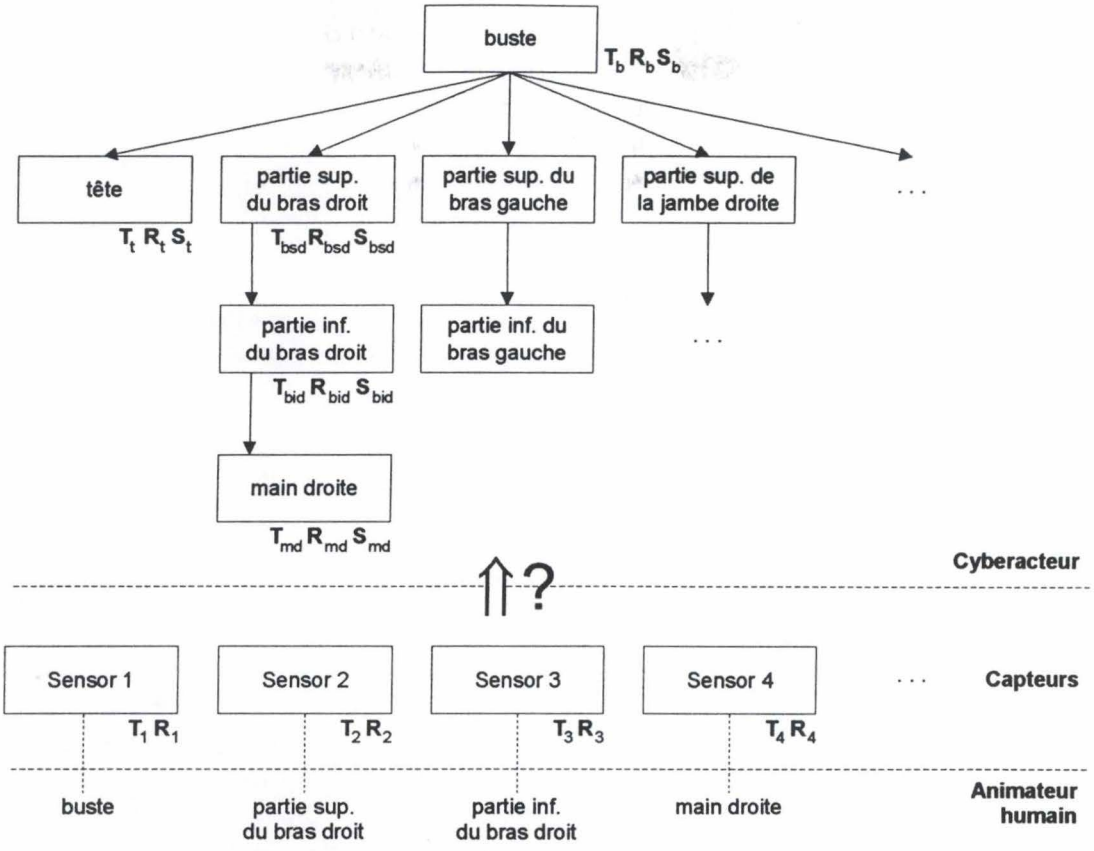


Figure 4.17 Comment appliquer les mouvements d'un être humain sur un cyberacteur ?

translation par celles du capteur auquel il est associé, pour que le modèle adopte les mêmes position et orientation globales que l'acteur humain. Sur l'exemple de la figure 4.17, cela donne :

$\mathbf{T}_b \leftarrow \mathbf{T}_1$
$\mathbf{R}_b \leftarrow \mathbf{R}_1$

$\mathbf{T}_1$ et $\mathbf{R}_1$ sont cependant exprimés dans le *frame* global *F*, tandis que $\mathbf{T}_b$ et $\mathbf{R}_b$ sont exprimés dans le *frame* global du modèle. Il est donc nécessaire de disposer de deux **matrices d'ajustement** $\mathbf{T}_{aj}$ et $\mathbf{R}_{aj}$ qui permettent d'exprimer les différences qu'il peut exister entre les deux *frames*. Les expressions ci-dessus deviennent alors :

$\mathbf{T}_b \leftarrow \mathbf{T}_{aj}\mathbf{T}_1$
$\mathbf{R}_b \leftarrow \mathbf{R}_{aj}\mathbf{R}_1$

Pour déterminer les modifications à réaliser sur les autres noeuds du modèle, prenons l'exemple du noeud «partie inf. du bras droit» de la figure 4.17. Les matrices $\mathbf{T}_{bid}$ et $\mathbf{S}_{bid}$ ne doivent pas être modifiées, sous peine de démembrer ou de redimensionner l'instance d'objet. Il nous reste $\mathbf{R}_{bid}$. Comment le modifier ? Le noeud «partie inf. du bras droit» est associé au capteur 3 attaché à la partie inférieure du bras droit de l'animateur humain : $\mathbf{R}_3$ détermine donc l'orientation de ce membre, dans *F*. En vertu de la définition de la matrice d'ajustement $\mathbf{R}_{aj}$, $\mathbf{R}_{aj}\mathbf{R}_3$ détermine l'orientation de ce membre dans le *frame* global du modèle. Intuitivement, il suffit de remplacer la matrice globale qui détermine l'orientation de la partie inférieure du bras gauche dans le cyberacteur, par $\mathbf{R}_{aj}\mathbf{R}_3$. Cependant, cette matrice globale n'existe pas telle quelle dans notre modèle : elle est donnée par

$\mathbf{R}_b\mathbf{R}_{bsd}\mathbf{R}_{bid}$

L'idée est donc d'exprimer $\mathbf{R}_{aj}\mathbf{R}_3$ localement, par rapport au noeud «partie supérieure du bras droit» du cyberacteur, et de remplacer $\mathbf{R}_{bid}$ par cette nouvelle représentation locale de $\mathbf{R}_{aj}\mathbf{R}_3$. Pour ce faire, on utilise l'égalité

$$\mathbf{R}_b\mathbf{R}_{bsd}\mathbf{R}_{bid} = \mathbf{R}_{aj}\mathbf{R}_3$$

On peut l'écrire sous la forme

$$\mathbf{R}_{acc.bsd}\mathbf{R}_{bid} = \mathbf{R}_{aj}\mathbf{R}_3 \qquad \text{avec } \mathbf{R}_{acc.bsd} = \mathbf{R}_b\mathbf{R}_{bsd}$$

$\mathbf{R}_{acc.bsd}$ est appelée la **matrice d'accumulation** en rotation du noeud «partie supérieure du bras droit». La matrice d'accumulation en rotation d'un noeud détermine son orientation par rapport au *frame* global du modèle. L'expression ci-dessus peut se réécrire

$$\mathbf{R}_{bid} = \mathbf{R}^{-1}_{acc.bsd}\mathbf{R}_{aj}\mathbf{R}_3$$

Nous allons donc remplacer $\mathbf{R}_{bid}$ par ce produit de matrices :

$$\mathbf{R}_{bid} \leftarrow \mathbf{R}^{-1}_{acc.bsd}\mathbf{R}_{aj}\mathbf{R}_3$$

De manière générale, la mise à jour de la matrice de rotation $\mathbf{R}_i$ d'un noeud *i* qui n'est pas la racine et auquel est associé un capteur *k*, est donnée par

$$\mathbf{R}_i \leftarrow \mathbf{R}^{-1}_{acc.père\ de\ i}\mathbf{R}_{aj}\mathbf{R}_k$$

$\mathbf{R}^{-1}_{acc.père\ de\ i}$ est la matrice d'accumulation en rotation du noeud père de *i* ; $\mathbf{R}_{aj}$ est la matrice d'ajustement entre *F* et le *frame* du modèle ; $\mathbf{R}_k$ donne l'orientation du capteur *k* dans *F*.

On voit que, dans cette solution de base, nous n'avons pas besoin des matrices de translation des capteurs, mise à part celle du capteur qui est associé à la racine du cyberacteur.

4.3.2. Matrices de calibration

Les capteurs qui sont fixés sur un être humain sont positionnés et orientés de manière approximative. En outre, chaque fois qu'un personne remet les capteurs après les avoir enlevés, ceux-ci ne sont jamais fixés exactement comme la première fois. Il est donc nécessaire de pouvoir «corriger» les matrices de translation et de rotation qui proviennent des capteurs. On peut le faire en utilisant des **matrices de calibration :**

• pour le capteur qui est associé au noeud racine, on définit deux matrices de calibration: une pour corriger la translation et une autre pour corriger la rotation. Dans l'exemple de la figure 4.17, si on utilise ces matrices, les matrices $\mathbf{T}_b$ et $\mathbf{R}_b$ deviennent

$$\mathbf{T}_b \leftarrow \mathbf{T}_{calib1}\mathbf{T}_{aj}\mathbf{T}_1$$
$$\mathbf{R}_b \leftarrow \mathbf{R}_{calib1}\mathbf{R}_{aj}\mathbf{R}_1$$

$\mathbf{T}_{calib1}$ et $\mathbf{R}_{calib1}$ sont les matrices de calibration en translation et en rotation du capteur 1.

• Pour chaque capteur associé à un noeud qui n'est pas la racine de l'arbre, on définit une matrice de calibration en rotation. En tenant compte de cette matrice, la mise à jour de la matrice de rotation $\mathbf{R}_i$ d'un noeud *i* qui n'est pas la racine et auquel est associé un capteur *k*, est donnée par

$$\mathbf{R}_i \leftarrow \mathbf{R}^{-1}_{acc.père\ de\ i}\mathbf{R}_{calibk}\mathbf{R}_{aj}\mathbf{R}_k$$

$\mathbf{R}_{calibk}$ est la matrice de calibration en rotation du capteur *k*.

Le contenu des matrices de calibration n'est pas déterminé. Dans l'implémentation de GabbyCapture, c'est l'utilisateur qui, par essais-erreurs, modifiera le contenu de ces matrices jusqu'à ce que le résultat visuel soit satisfaisant.

4.3.3. Problèmes non résolus

Les sections 4.3.1 et 4.3.2 exposent la solution telle qu'elle sera implémentée par GabbyCapture. Cette solution constitue une solution de base, qui ne prend pas en charge tous les problèmes qui sont liés à la capture de mouvements.

Le problème principal qui reste à résoudre est celui des différences de morphologie qu'il peut exister entre l'animateur humain et le modèle 3D. Imaginons qu'un humain maigre mette sa main à plat sur son torse, et que ce mouvement doive être appliqué sur un gros cyberacteur. La main du cyberacteur risque de passer au travers de son propre torse... Une solution à ce problème serait de définir les angles minimum et maximum qui peuvent être déterminés par les articulations du cyberacteur et de l'humain, et d'adapter les angles formés par les articulations de l'être humain en fonction des angles que les articulations du cyberacteur peuvent supporter.

D'autres problèmes sont liés aux différences de morphologie : l'adaptation de la vitesse de rotation des membres qui, sur l'être humain, ont une taille différente que ceux qui lui correspondent sur le modèle, le problème des pieds du cyberacteur devant toujours rester sur le «sol» de son espace virtuel... Ces problèmes ne seront pas abordés dans le cadre de ce travail.

Chapitre 5

Architecture de GabbyCapture

La spécification présentée au chapitre précédent exposait notre problème de capture de mouvements sans se soucier de la manière dont sa solution allait prendre forme en termes informatiques. Le chapitre 5 présente une solution informatique à ce problème, qui a été réalisée après GabbyCapture et aussi après la spécification du chapitre 4. Cette solution a été conçue, comme la spécification, dans le but de proposer une démarche de développement alternative à celle que nous avons adoptée pour l'écriture de GabbyCapture.

On peut se demander de quelle manière cette solution —qu'on appelera aussi architecture— a été conçue : l'a-t-elle été uniquement sur base de la spécification, ignorant tout à fait le code qui avait déjà été écrit, ou, au contraire, s'est-elle basée entièrement sur ce code, s'efforçant de réaliser la correspondance la plus exacte possible entre les concepts du code et ceux, plus abstraits, du langage avec lequel elle a été écrite?

Au départ, cette architecture a été écrite avec la volonté d'être fidèle au code de GabbyCapture. Au fur et à mesure de son écriture, cependant, quelques parties ont été réécrites sans plus tenir compte de la manière dont le code avait été conçu. L'architecture qui a résulté de ce processus est «hybride»: elle ne correspond pas parfaitement au code de GabbyCapture mais ne correspond pas non plus à la solution que j'aurais écrite si j'étais reparti de rien, si ce n'est de la spécification.

Le processus mis en évidence dans ce chapitre est celui qui a consisté à écrire l'architecture d'un logiciel, GabbyCapture, dont, d'une part, le code avait déjà été écrit, et qui, d'autre part, avait aussi été repensé depuis le début, par la réalisation de sa spécification. Ce processus a permis de confronter deux démarches de développement : celle qui consiste à écrire directement le code et celle qui consiste à réaliser une spécification et une architecture avant d'écrire le code. Cette confrontation a pu être établie parce que le processus a produit une architecture qui, tout en restant très proche du code de GabbyCapture, a pu être évaluée avec un autre regard, engendré par le fait d'avoir adopté une démarche de développement *top-down*, débutant par une définition précise du problème. Cette autre démarche a d'ailleurs fait naître une série d'idées concernant les améliorations que l'on pourrait apporter à l'architecture de GabbyCapture.

Dans ce chapitre, nous commencerons par présenter le langage avec lequel l'architecture de Gabby a été écrite —le langage ROOM— et nous donnerons les raisons pour lesquelles nous l'avons choisi (5.1). Nous présenterons ensuite l'architecture en elle-même (5.2) ; on trouvera sa description complète en ROOM dans l'annexe A1 (à partir de la page 91). Le point 5.3 évalue l'architecture ROOM en critiquant et en justifiant les choix qui ont été effectués tout au long de sa réalisation, et en proposant parfois des solutions alternatives. Pour les parties de l'architecture qui correspondent au code de GabbyCapture (et qui sont majoritaires), ces critiques et justifications sont donc celles des choix effectués lors de l'écriture du code [1]. Pour les parties qui ne correspondent pas au code, on exposera et on justifiera le code qui avait été écrit dans GabbyCapture et on le comparera aux modifications qui ont été apportées.

L'évaluation de notre architecture ROOM, de par la nature de celle-ci, correspond à une comparaison entre les deux démarches de développement qui ont été mises en présence l'une avec l'autre. L'objectif est de montrer les avantages que peuvent apporter l'adoption d'une démarche de développement *top-down*, depuis la spécification du problème jusqu'à son implémentation en passant par son architecture, en utilisant un outil d'analyse spécifique, disposant de concepts de haut niveau, comme ROOM.

5.1. Le langage ROOM

Notre architecture a été écrite avec le langage ROOM (Real-time Object-Oriented Modeling).

ROOM a été conçu pour dans le but de modéliser un type de systèmes particulier : les systèmes réactifs, concurrents, temps-réel. Ses concepteurs sont partis du constat suivant : *les architectures logicielles sont souvent définies de manière informelle, en utilisant en général plusieurs types de représentations différentes accompagnées de commentaires textuels. Par conséquent, ces architectures sont souvent mal comprises, peu utilisées ou même complètement ignorées.* [...] *Il n'est donc pas du tout garanti que les implémentations des systèmes décrits de cette manière correspondront à ce qui était voulu au départ.* [...] *De plus, lors de la maintenance du système, l'écart peut encore se creuser entre ces architectures et leurs implémentations, créant ainsi ce que nous appelons un «déclin architectural»* ([ObjecTime]*).

Pour éviter ce déclin, ROOM a été conçu comme un ensemble intégré de concepts formels qui, avec les outils appropriés, sont traduisibles à 100% en code exécutable. La marge entre l'architecture et le code devient ainsi très réduite [2] : une partie importante de la maintenance peut directement être réalisée sur l'architecture, garantissant ainsi un «déclin» minimal.

Le concept fondamental, dans ROOM, est celui d'*acteur* : il s'agit d'un objet ayant son propre «chemin d'exécution». Chaque acteur dispose de «portes» qui sont reliées aux portes d'autres acteurs et qui permettent à tous ces acteurs de communiquer, selon des protocoles déterminés. Ces liaisons sont des liaisons point-à-point, reliant uniquement une porte à une autre. Un acteur peut contenir d'autre acteurs.

[1] Puisque nous utilisons, pour décrire GabbyCapture, un langage plus abstrait que celui dans lequel le code de GabbyCapture a été écrit, nous n'exposerons ni ne critiquerons les choix qui concernent les «détails d'implémentation» masqués par ROOM.

[2] ROOM ne permet pas de modéliser l'interface utilisateur d'un système. Les possibilités qu'il offre au niveau de la modélisation des données ne sont pas très développées non plus, ce qui s'explique par le fait qu'il a pour but de modéliser des systèmes pour lesquels la rapidité est indispensable.

Le comportement de chaque acteur est décrit par un diagramme états-transitions. En règle générale, un acteur franchit une transition dès qu'il reçoit un message sur une de ses portes, pour autant qu'une certaine condition soit remplie. Les actions que l'acteur peut exécuter lorsqu'il franchit une transition sont l'envoi d'autres messages via ses portes et/ou l'exécution d'actions internes. Ces actions sont écrites dans un langage de programmation comme C++ ou JAVA [1].

La figure 5.1 montre un exemple de modèle ROOM.

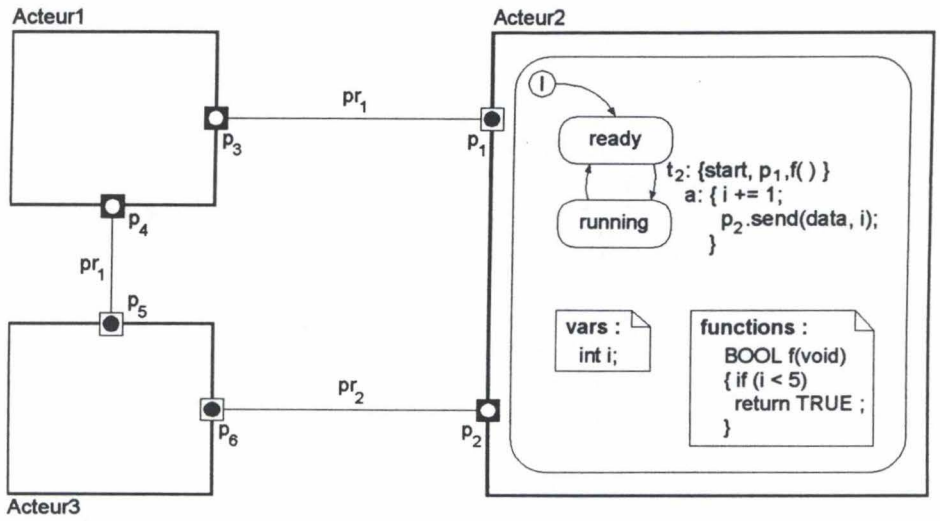


Figure 5.1 Un exemple de modèle ROOM

Sur cette figure, l'acteur Acteur2 dispose d'une porte p_1 lui permettant de communiquer avec l'acteur Acteur1 via la porte p_3 de celui-ci selon un certain protocole pr_1. L'acteur Acteur2 peut également communiquer avec l'acteur Acteur3 via sa porte p_2. Acteur2 peut se trouver dans deux états différents : ready et running. La lettre 'I' entourée d'un cercle à partir duquel une flèche est dessinée jusqu'à l'état ready indique que celui-ci est l'état initial de l'acteur. La transition t_2 de Acteur2 est franchie et fait passer cet acteur de ready à running s'il reçoit un message de type {start} sur sa porte p_1 et si la fonction f, qui est une fonction définie en C++, renvoie TRUE. Quand Acteur2 franchit cette transition t_2, il incrémente la variable C++ i de 1 et envoie à Acteur3, via sa porte p_2, un message de type {data, int} auquel il associe une valeur entière qui est la valeur de i. Le type de message {data, int} doit avoir été défini dans le protocole pr_2 ; le type de message {start} doit avoir été défini dans le protocole pr_1.

Une introduction générale à ROOM peut être trouvée dans [ObjecTime]*. On trouvera dans [Sel94] un exposé plus systématique.

Pourquoi avoir choisi le langage ROOM pour écrire l'architecture de GabbyCapture ? Pour les quatre raisons suivantes.

1• ROOM a été choisi d'abord parce que c'est un langage orienté objets, qui offre donc des possiblités intéressantes au niveau de la structuration, de la réutilisabilité, de la maintenabilité,

[1] Les actions de notre modèle ont été écrites avec C++.

etc. Nous reviendrons sur ces caractéristiques par la suite.

2• ROOM a aussi été choisi parce que c'est un langage qui est bien adapté au problème que nous avons à résoudre. Bien que les contraintes temps-réel qu'il est possible de définir dans ROOM[1] permettent de minuter le déclenchement des opérations, ce qui aurait pu être utile pour afficher, 25 fois par seconde, un cyberacteur animé à l'écran, ces contraintes n'ont pas été utilisées. En effet, pour pouvoir appliquer ces contraintes, l'OS sur lequel tourne le modèle ROOM doit pouvoir garantir les temps de réponse demandés : il doit donc être un OS temps-réel, ce qui n'était pas le cas de la version de Unix sur laquelle tourne GabbyCapture. Puisque le système MotionStar envoie par exemple 25 messages par seconde, la démarche qui a été adoptée lors de l'écriture du code a été de minimiser le temps de traitement de chaque message, de manière à ce que tous les messages provenant du système de capteur puissent être traités.

Pour revenir à ROOM, on peut dire qu'il a été adapté pour modéliser les aspects de concurrence et de «réactivité» inhérents au problème de la capture de mouvements. En effet, les messages qui proviennent de l'environnement de GabbyCapture sont nombreux et doivent être traités le plus rapidement possible. Ils proviennent soit du système de capteurs, soit de l'utilisateur, qui peut, pendant l'animation du cyberacteur, calibrer les mouvements de celui-ci ou modifier la configuration du système de capteurs pour activer ou désactiver le flux de données provenant de tel ou tel capteur. Au niveau du code de GabbyCapture, beaucoup de mécanismes ont été mis en oeuvre pour traiter ces aspects : processus, boucles d'évenements, mécanismes de communication inter-processus (mémoire partagée, sockets TCP, signaux Unix...), etc. Le langage de ROOM permet de traiter ces problèmes de manière uniforme, avec un minimum de concepts.

3• ROOM a également été choisi parce qu'il s'agit d'un langage formel de «haut niveau». Cette caractéristique fournit un double avantage : non seulement il est possible d'exprimer très précisément une solution informatique sans avoir à considérer une myriade de détails techniques sans réelle importance —comme ce serait le cas en utilisant C ou C++— mais il est également possible, en ayant les outils adéquats, d'obtenir du code exécutable.

4• ROOM a enfin été choisi parce qu'il ne m'était pas complètement inconnu : j'ai en effet réalisé quelques travaux sur ce langage ainsi que sur l'outil qui le supporte (ObjecTime), dans le cadre des cours de méthodologie de développement de logiciels d'Eric Dubois.

D'autres outils d'analyse temps-réel existent. Certains adoptent une approche sur base de machines à états finis, comme ROOM avec ses *ROOMCharts*, ou StateMate avec ses *statecharts* dont ROOM s'est inspiré. D'autres, par contre, utilisent les réseaux de Petri pour modéliser le comportement des systèmes temps-réel. On peut trouver dans [Buc95] une synthèse des différents outils et notations utilisées pour spécifier ce type de systèmes.

Nous n'avons pas choisi ROOM après l'avoir évalué par rapport à d'autres outils, pour la simple raison qu'il nous aurait été impossible, en terme de temps, de modéliser GabbyCapture avec un langage que nous ne connaissions pas au préalable.

[1] Tout acteur ROOM peut envoyer des messages à un acteur prédéfini —le *Timing Actor*— pour initialiser des *timers*. Ces messages sont de la forme «préviens-moi dans exactement 40 millisecondes». Le Timing Actor, quand le temps s'est écoulé, renvoie un message à l'acteur concerné. Celui-ci peut donc minuter précisément le déclenchement de ses actions.

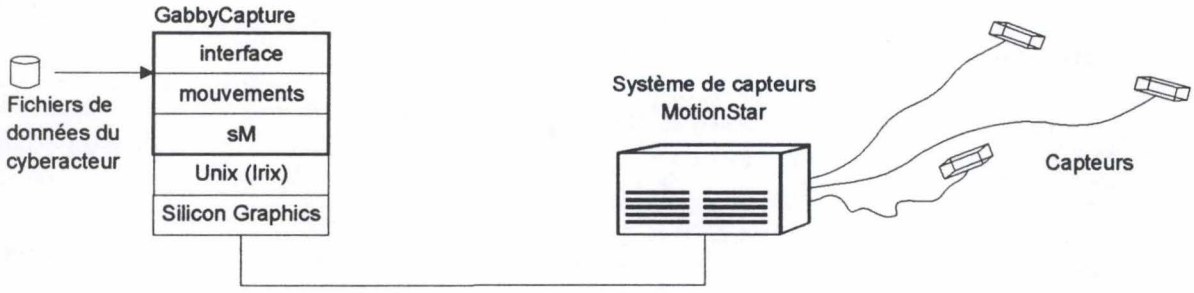
5.2. Architecture ROOM de GabbyCapture

Cette section contient une présentation générale de l'architecture que nous avons réalisée. La description ROOM complète fait l'objet de l'annexe A1 (pages 91 et suivantes). Ici, nous proposerons d'abord une vue d'ensemble de l'architecture (5.2.1), puis nous reviendrons plus en détail sur trois éléments de celle-ci (5.2.2 à 5.2.4).

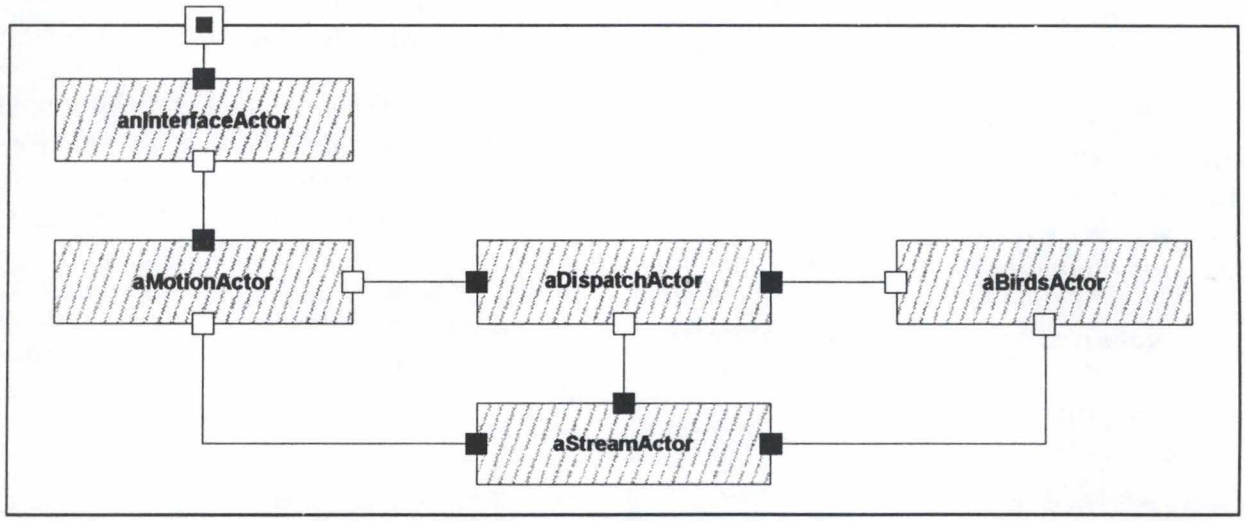
5.2.1. Vue d'ensemble

Nous avons choisi, dans cette architecture, de modéliser le système de capture de mouvements entier, qui comprend GabbyCapture mais aussi le système de capteurs MotionStar, afin d'avoir une vue homogène de l'ensemble de ce système. L'interface de GabbyCapture est donc également présente dans notre modèle, et ce pour la même raison, bien que ROOM ne permette pas de modéliser des interfaces. Dans l'annexe A1, nous ne décrirons donc pas de manière formelle les acteurs qui correspondent au système de capteurs et à l'interface de GabbyCapture.

La figure 5.2 reprend la représentation de GabbyCapture et du système de capteurs MotionStar



(a) Représentation du chapitre 3



(b) Représentation sous la forme d'acteurs ROOM

Figure 5.2 Le système de capture de mouvements

telle qu'elle était exposée dans le chapitre 3 et présente aussi leur représentation sous la forme d'acteurs ROOM [1].

Sur cette figure, le système MotionStar correspond à l'acteur ROOM aBirdsActor [2]. Le module 'Sm', responsable de la «récupération» des données en provenance du système de capteurs, correspond aux acteurs aDispatchActor et aStreamActor. Le module 'mouvements', qui, sur base du cyberacteur chargé à partir des fichiers Gabby et sur base des données provenant de 'Sm', calcule la posture du cyberacteur, correspond à aMotionActor. Le module 'interface', quant à lui, correspond à anInterfaceActor.

aMotionActor est l'acteur central du modèle ROOM. Il contient deux structures de données importantes : la première, Model, est utilisée pour contenir le cyberacteur, et la seconde, Sensors, comprend les mesures qui sont générées par les capteurs. Chaque fois qu'un message de données lui provient de aStreamActor (message contenant les mesures de tous les capteurs à un instant donné), aMotionActor met à jour la structure Sensors sur base de ce message, met ensuite à jour Model sur base de Sensors puis envoie les messages nécessaires à anInterfaceActor pour que celui-ci puisse afficher le cyberacteur à l'écran.

anInterfaceActor, d'une part, reçoit de aMotionActor des messages qui concernent l'affichage du cyberacteur, et, d'autre part, reçoit les commandes de l'utilisateur.

- Quand l'utilisateur demande de charger un cyberacteur, anInterfaceActor envoie les messages nécessaires pour que aMotionActor initialise sa structure Model ;
- Quand l'utilisateur associe un membre du cyberacteur avec un capteur, anInterfaceActor envoie les messages nécessaires à aMotionActor pour que celui-ci modifie ses structures Model et Sensors en conséquence ;
- Quand l'utilisateur demande de modifier la configuration du système de capteurs, anInterfaceActor transmet le message à aMotionActor [3].
- L'utilisateur peut également lancer ou stopper l'animation du cyberacteur et calibrer les mouvements de celui-ci.

aBirdsActor, dès que aStreamActor lui en donne l'ordre, mesure, à un rythme de Nm ($20 \leq Nm \leq 90$) fois par seconde, la position et/ou l'orientation de chacun de ses capteurs, ou encore aucune des deux.

En effet, aBirdsActor permet de configurer le type des données qui sont envoyées par chaque capteur. Pour animer un cyberacteur entier avec notre solution du chapitre 4, nous avons vu que nous avions besoin de l'orientation de tous les capteurs et que nous pouvions nous passer de leur position, mise à part celle du capteur associé au noeud racine du cyberacteur. Dans le chapitre 3, nous avons vu que pour calibrer une articulation d'un cyberacteur, la «désactivation» des autres capteurs pouvait être utile. Enfin, il se peut que nous n'ayons pas besoin d'enregistrer les mouvements de l'entièreté du corps du cyberacteur. Il est donc nécessaire de disposer d'un maximum de souplesse dans la configuration des données générées par chaque capteur. Les formats de données possibles sont les suivants : aucune donnée, position du capteur uniquement, orientation uniquement, position et orientation.

[1] La représentation sous forme d'acteurs ROOM correspond à la représentation de la classe MotionCaptureSystem (annexe A1, page 92), dont la seule instance est le système de capture de mouvements en entier.

[2] L'ancien nom du système MotionStar, qui apparaît un peu partout dans la documentation du système, est Flock of Birds, d'où le nom de aBirdsActor.

[3] Nous avons vu au chapitre 3 que l'utilisateur pouvait, par exemple, «activer» ou «désactiver» le flux de données provenant d'un capteur particulier.

La position et l'orientation d'un capteur sont exprimés dans un *frame* global défini dans aBirdsActor. Nm fois par seconde, celui-ci rassemble dans un message l'ensemble des mesures concernant tous les capteurs à un instant donné et envoie ce message à aStreamActor. aBirdsActor reçoit également les ordres de modification de configuration qui lui proviennent de aDispatchActor.

Outre le type des données émises par chaque capteur, ces ordres de modification peuvent aussi concerner le taux Nm auquel le système MotionStar envoie ses mesures. Il est typiquement de 25 fois par seconde.

aStreamActor «formate» les messages de données qu'il reçoit de aBirdsActor pour que ces données soient utilisables par aMotionActor. Il envoie d'ailleurs les résultats de ses formatages à ce dernier. aStreamActor envoie à aBirdsActor l'ordre d'envoyer des messages de données dès qu'il en reçoit lui-même l'ordre de aDispatchActor.

aDispatchActor relaye les messages qui proviennent de aMotionActor vers aBirdsActor et aStreamActor. Par exemple, quand aMotionActor lui enjoint de lancer ou de stopper l'animation, aDispatchActor transmet ces ordres à aStreamActor. Quand aMotionActor lui envoie une nouvelle configuration, aDispatchActor la transmet à aStreamActor et à aBirdsActor. aDispatchActor joue aussi un rôle important en ce qui concerne l'initialisation et la clôture du système de capture de mouvements.

Les différents messages que s'envoient les acteurs sont formalisés dans des classes de protocoles que l'on peut retrouver dans l'annexe A1, à partir de la page 93.

Les 3 points suivants abordent plus en détail les acteurs aMotionActor, anInterfaceActor et aDispatchActor. On les retrouvera aussi dans l'annexe A1, respectivement aux pages 98, 111 et 129. aBirdsActor et aStreamActor, qu'on ne définira pas plus dans ce chapitre, sont également définis dans l'annexe A1 aux pages 120 et 124.

5.2.2. *'aMotionActor'*

aMotionActor contient les structures de données Model et Sensors, accueillant respectivement le cyberacteur et un certain nombre d'informations concernant les capteurs. Ces structures prennent la forme de deux instances de classes C++.

- Model est un arbre à chaque noeud duquel sont associées :
  - deux matrices déterminant respectivement la position et l'orientation de l'instance d'objet associée à ce noeud par rapport à l'instance de son noeud père,
  - une référence vers l'objet-symbole (avant-bras, torse, tête...) de l'instance associée à ce noeud ;
  - une matrice d'accumulation, qui détermine la position et l'orientation de l'instance associée au noeud par rapport au *frame* global du cyberacteur.

Les sommets qui définissent les objets-symboles sont stockés dans des structures spécialement conçues pour l'affichage de données graphiques. Ces structures sont appelées *display lists*, sont définies dans la bibliothèque OpenGL et sont contenues, dans notre design, dans l'acteur anInterfaceActor. On peut trouver dans [OpenGL]* ou [Nei93] une description des fonctionnalités d'OpenGL, qu'on retrouvera souvent dans notre design.

- Sensors contient des informations relatives aux capteurs :

  - leur nombre,
  - le nombre de messages de données envoyés par seconde par les capteurs,
  - deux matrices, pour chaque capteur, déterminant respectivement la position et l'orientation du capteur par rapport au *frame* global de aBirdsActor ;
  - deux matrices de calibration pour chaque capteur : une en translation et une en rotation. La matrice de calibration en translation n'est utilisée que pour le capteur attaché au noeud racine du cyberacteur.

Les définitions C++ de ces deux objets et des classes dont ils sont les instances sont présentées et commentées dans l'annexe A1 à partir de la page 100.

Model est initialisé à partir d'un ensemble de fichiers texte contenus dans un volume quelconque, lui-même présent sur un (ou plusieurs) support(s) physique(s). Ce volume n'est pas représenté sur la figure 5.2 (b) ; il est resté, dans notre modèle, sous la forme de code C++. Ce volume pourrait néanmoins constituer un sixième acteur ROOM : aMotionActor communiquerait avec cette instance —appelons-la aVolume— pour obtenir le contenu des fichiers texte à partir desquels il initialise Model. La figure 5.3 ci-dessous présente un modèle ROOM qui modélise l'instance d'acteur aVolume.

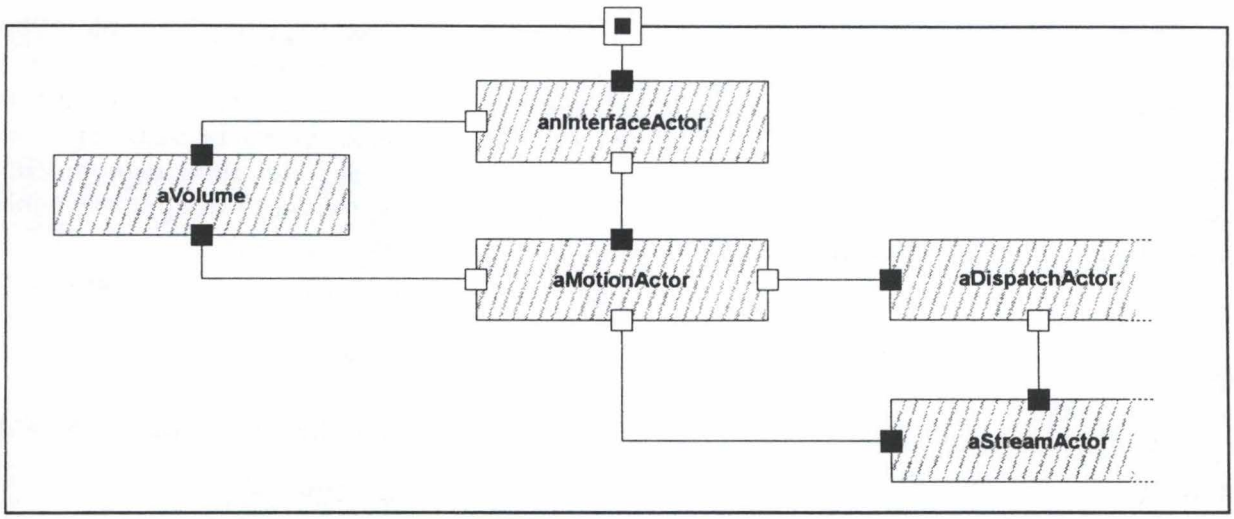


Figure 5.3 Le système de capture de mouvements avec aVolume

Lors de l'initialisation du système, aMotionActor reçoit de aDispatchActor le nombre de capteurs ainsi que le nombre de messages que ceux-ci envoient par seconde, et initialise sa structure Sensors sur base de ces informations. Les matrices de Sensors déterminant la position et l'orientation de chaque capteur sont mises à jour sur base des messages qui sont envoyés Nm fois par seconde par aStreamActor. Les matrices de calibration sont bien sûr initialisées et modifiées par l'utilisateur du système, et sont donc envoyées à aMotionActor par l'acteur anInterfaceActor.

L'utilisateur peut décider d'associer un à un les capteurs et les membres du personnage 3D (seule limitation : le premier capteur est assigné obligatoirement au noeud racine du cyberacteur). A chaque fois qu'une association est demandée, anInterfaceActor envoie la demande à aMotionActor, qui modifie Model et Sensors de manière à refléter cette association.

Dès que aMotionActor reçoit, de anInterfaceActor, l'ordre de lancer l'animation du cyberacteur, il le transmet à aDispatchActor qui fait le nécessaire pour que aStreamActor envoie Nm fois par seconde les messages de données à aMotionActor. Celui-ci, après avoir mis à jour, Nm fois par seconde, les matrices de translation et/ou de rotation de chaque capteur de Sensors sur base des messages en provenance de aStreamActor, met à jour Model sur base des données de Sensors pour faire adopter au modèle 3D la même position et la même posture que l'être humain auquel sont attachés les capteurs.

En termes plus précis, Nm fois par seconde :

- la matrice déterminant la position du noeud racine est remplacée par le produit de la matrice déterminant la position du capteur qui lui est associé par la matrice de calibration en translation de ce capteur ;
- la matrice déterminant l'orientation du noeud racine est remplacée par le produit de la matrice déterminant l'orientation du capteur qui lui est associé par la matrice de calibration en rotation de ce capteur ;
- pour chaque noeud auquel un capteur est associé, la matrice déterminant l'orientation de l'instance associée à ce noeud est remplacée par un produit de matrices incluant la matrice d'accumulation en rotation de son noeud père, inversée, la matrice déterminant l'orientation du capteur attaché au noeud ainsi que la matrice de calibration en rotation de ce capteur.

A chaque fois que aMotionActor met à jour Model de cette manière, il envoie à anInterfaceActor les messages nécessaires pour que celui-ci affiche le modèle dans une de ses fenêtres. Un premier résultat est alors atteint : le modèle est animé à l'écran en fonction des mouvements de l'être humain sur lequel sont fixés les capteurs. A partir du moment où l'utilisateur voit le résultat, il peut, via une fenêtre de anInterfaceActor, modifier les valeurs des matrices de calibration des capteurs pour optimiser le résultat. Bien entendu, anInterfaceActor envoie ces nouvelles valeurs de calibration à aMotionActor qui en tient compte dans sa mise à jour du modèle.

5.2.3. *'anInterfaceActor'*

anInterfaceActor représente l'interface utilisateur du système de capture de mouvements et contient donc un certain nombre d'«objets interactifs» :

- un menu principal ;

- une fenêtre browseWin qui permet de parcourir le catalogue du volume aVolume dont nous parlions plus haut. Quand l'utilisateur appuie sur l'item '*open*' du menu '*file*' du menu principal, anInterfaceActor demande à aVolume son catalogue et affiche browseWin avec le catalogue de aVolume. Quand l'utilisateur a sélectionné dans le catalogue le nom du fichier texte principal correspondant au modèle à charger et qu'il appuie sur le bouton '*Ok*' de browseWin, anInterfaceActor envoie le nom de ce fichier à aMotionActor ainsi que l'ordre d'initialiser son acteur Model. aMotionActor initialise celui-ci en communiquant avec aVolume. Comme nous le disions plus haut, celui-ci est resté dans notre modèle sous la forme de code C++ mais est représenté sous la forme d'acteur sur la figure 5.3.

- anInterfaceActor comprend une fenêtre openGLWin dans laquelle le modèle 3D est affiché.

- une fenêtre attdetWin permet à l'utilisateur d'associer un à un (ou de dissocier) les capteurs

et les membres du modèle 3D. Les modifications d'associations sont communiquées à aMotionActor.

• une fenêtre configWin permet à l'utilisateur de modifier la configuration de l'acteur aBirdsActor : nombre de messages que les capteurs envoient par seconde et format des données envoyées par chaque capteur. Les modifications de configuration sont envoyées par anInterfaceActor à aMotionActor qui les transmet à aBirdsActor par l'intermédiaire de aDispatchActor.

• anInterfaceActor comprend encore une fenêtre calibWin qui fournit un moyen à l'utilisateur de modifier les matrices de calibration en translation et en rotation de chaque capteur. Ces modifications sont signalées à aMotionActor.

• Une boîte de dialogue controlsWin, toujours visible, permet à l'utilisateur :
  • de contrôler l'animation (boutons ' *start* ' et ' *stop* '),
  • de manipuler une «caméra virtuelle» (rotations et translations en x, y et z) qui est pointée sur le modèle dans la fenêtre openGLWin pour pouvoir observer le modèle avec des points de vue différents.

La figure 5.4 montre la représentation graphique de anInterfaceActor, du menu principal, des fenêtres controlsWin, openGLWin et attdetWin. Un cyberacteur-test (Oscar) est affiché dans openGLWin. Les cubes verts dessinés sur certains de ses membres signalent que des capteurs y sont associés.

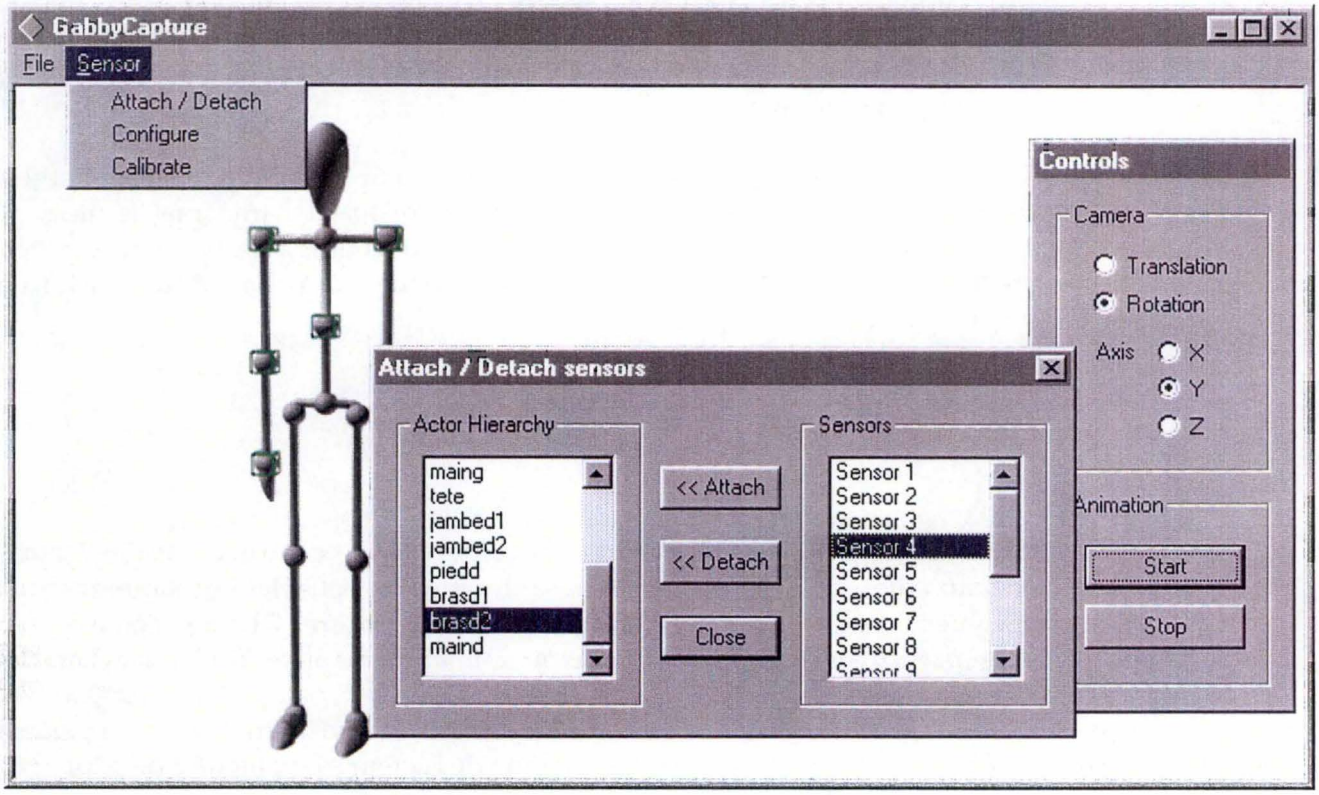


Figure 5.4
Représentation graphique de anInterfaceActor, du menu principal, d'openGLWin, de attdetWin et de controlsWin

5.2.4. *'aDispatchActor'*

aDispatchActor relaye les messages qui proviennent de aMotionActor vers aBirdsActor et aStreamActor et joue aussi un rôle en ce qui concerne l'initialisation et la clôture du système de capture de mouvements.

Initialisation • Lors de la «création» du système de capture de mouvements, les 5 acteurs de la figure 5.2 (b) ne sont pas tous créés ensemble. Ceci est dû à l'implémentation des acteurs: aBirdsActor correspond, comme nous l'avons déjà dit, au système de capteurs MotionStar, qui est un PC 486 doté des périphériques nécessaires (dont les capteurs), tandis que les 4 autres acteurs se retrouvent sous la forme de 3 processus tournant sur une machine Unix. Deux cas de figure peuvent dès lors se produire lors de l'initialisation :

- le 486 est lancé avant le programme (les 3 processus) sous Unix. Dans ce cas, on considère que le système est créé, et que seul aBirdsActor est « incarné » (dans le jargon ROOM). Les 4 autres, puisqu'ils n'existent pas encore à ce moment, n'existent donc pas pendant *toute* la durée de vie du système et c'est la raison pour laquelle ils sont représentés sur la figure 5.2 comme étant des acteurs «optionnels» (hachurés). Quand le programme Unix est lancé, aDispatchActor est incarné à son tour et ce n'est qu'après s'être assuré que aBirdsActor était en activité (en lui envoyant un message de connexion) que aDispatchActor crée aStreamActor et aMotionActor, ce dernier créant à son tour anInterfaceActor.

- le programme Unix est lancé avant le 486. Dans ce cas, on considère le système est créé, et seul aDispatchActor est incarné. aBirdsActor doit donc aussi être optionnel. Dans ce cas-ci, aDispatchActor n'attend pas que aBirdsActor soit créé : il n'attend que 2 secondes puis se termine.

Clôture • La décision de clore le système vient de l'utilisateur via anInterfaceActor : celui-ci transmet le message à aMotionActor et se termine. aMotionActor transmet le message à aDispatchActor et fait de même ; aDispatchActor, avant d'en finir avec lui-même, transmet le message à aStreamActor et envoie un message de déconnexion à aBirdsActor qui, lui, reste en activité et doit être clôturé séparément.

5.3. Evaluation de l'architecture ROOM

Dans la présente section, nous allons tenter d'évaluer l'architecture de GabbyCapture, principalement selon deux critères. Le premier concerne sa structure : est-elle claire et compréhensible? Pourrait-on la faire évoluer sans devoir remettre en question son «ossature» ? Le deuxième concerne ses performances, et en particulier son aptitude à pouvoir animer de manière fluide un cyberacteur.

Nous n'allons bien sûr aborder que les acteurs qui constituent GabbyCapture : nous ne parlerons plus de aBirdsActor. Nous allons commencer par l'évaluation de l'acteur aDispatchActor (5.3.1). Nous nous demanderons ensuite quelle est l'efficacité de l'acheminement des mesures relatives aux capteurs, depuis leur génération par le système de capteurs jusqu'à la mise à jour du cyberacteur (5.3.2). Dans ce point, nous aborderons donc aStreamActor et aMotionActor. Dans un troisième temps, nous nous focaliserons sur l'arbre Model contenu dans aMotionActor et sur les différents parcours

d'arbre dont il fait l'objet (5.3.3). Nous terminerons en examinant certains aspects plus ciblés de l'architecture (5.3.4).

5.3.1. Evaluation de aDispatchActor

DispatchActor joue deux rôles dans le système de capture de mouvements : d'une part, il est chargé de relayer les messages en provenance de aMotionActor vers aBirdsActor et aStreamActor et, d'autre part, il initialise le système et prend part à sa clôture.

Le rôle que aDispatchActor joue en ce qui concerne l'initialisation et la clôture du système pourrait, moyennant des modifications mineures, être joué par n'importe quel autre acteur. Il ne s'agit pas d'un rôle important : il ne peut pas justifier à lui seul l'identification d'un acteur. Dès lors, le rôle de relais que joue aussi aDispatchActor justifie-t-il l'existence de celui-ci ?

L'identification d'un acteur n'a de sens que si elle permet de structurer le programme, de manière à ce qu'il soit plus compréhensible et qu'il reflète les caractéristiques du domaine d'application, ou bien si elle est nécessaire en terme de performance. Concernant aDispatchActor, nous allons voir que ces conditions ne sont pas remplies.

Des messages de 4 types peuvent provenir de aMotionActor : ils correspondent aux actions de l'utilisateur «lancer l'animation», «stopper l'animation», «quitter l'application» et «modifier la configuration du système». La nature de ces messages implique que le rythme auquel ils sont émis est lent. Quand il reçoit les messages «lancer» et «stopper l'animation», aDispatchActor ne fait rien, si ce n'est les transmettre à aStreamActor. Quand il reçoit les messages «quitter l'application» et «modifier la configuration», la seule chose qu'il fait est de dupliquer ces messages pour, à nouveau, les transmettre à aBirdsActor et à aStreamActor. Les autres types de messages traités par aDispatchActor sont peu nombreux et concernent uniquement l'initialisation du système.

Au niveau de la structure, aDispatchActor ne constitue pas une unité logique de fonctionnement. C'est un élément «creux» qui a été vidé de sa substance après une série de modifications concernant le code de GabbyCapture. La structure du programme est donc compliquée inutilement, rendant sa compréhension et sa maintenance plus difficile.

Au niveau de la performance, aDispatchActor ne constitue qu'un maillon inutile (ou presque) dans les chaînes de messages qui circulent dans le système. En effet, puisque aDispatchActor n'apporte aucune «valeur ajoutée» sur ces messages, aMotionActor pourrait se passer de ses services et envoyer ses messages directement à aStreamActor et aBirdsActor. On pourrait, par exemple, fusionner aDispatchActor avec aStreamActor. On aurait donc pu éviter qu'un acteur de plus soit exécuté en concurrence avec les autres. De plus, l'envoi de certains messages aurait pu se faire plus rapidement — bien qu'il ne s'agisse pas de messages «urgents», comme pourraient l'être les messages de données en provenance des capteurs.

Pourquoi aDispatchActor existe-t-il ? Initialement, aDispatchActor et aStreamActor ne devaient faire qu'un. L'implémentation du code qui équivaut à la machine à états finis ROOM de cet unique acteur posait quelques problèmes : nous avons donc décidé de le scinder en deux. Puisque nous n'évaluions pas les conséquences de tels changements sur l'ensemble du logiciel, rien ne s'opposait à ce que nous le fassions, d'autant plus que l'implémentation s'en trouvait facilitée. Ce n'est qu'en réalisant le modèle ROOM que nous nous sommes rendus compte de l'inutilité de cet acteur.

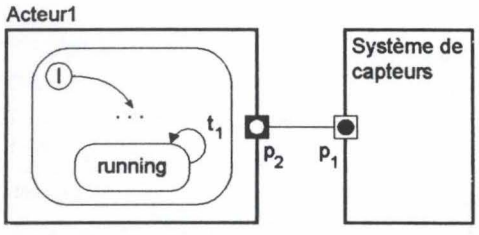
5.3.2. *Acheminement des données depuis le système de capteurs jusqu'au cyberacteur*

Le système de capteurs, quand l'animation du modèle est en cours, génère des messages de données à un certain taux, comme 25 messages par seconde. Un des éléments critiques, dans GabbyCapture, est de minimiser le temps qu'il faut pour traiter chacun de ces messages, c'est-à-dire le temps séparant l'émission du message de l'affichage du cyberacteur en fonction des données présentes dans le message.

D'après notre modèle ROOM, nous pouvons voir que 3 traitements sont nécessaires pour que les données provenant d'un tel message soient applicables au cyberacteur :

1• Puisque les données qui proviennent de **aBirdsActor** ne sont pas manipulables telles quelles par **aMotionActor** (la raison exacte est exposée dans l'annexe A1, page 124), il s'agit donc de leur appliquer un premier «formatage».
2• La structure de données **Sensors** (dans **aMotionActor**), contenant les matrices qui accueillent les données provenant des capteurs, doit être calculée ou remplie [1] sur base des données formatées.
3• L'arbre **Model** qui représente le cyberacteur doit être mis à jour sur base de **Sensors** et être affiché à l'écran. La mise à jour doit également tenir compte des «associations» qui le lient à cette structure de données, et qui déterminent quel capteur est attaché à quel membre du personnage.

Le temps séparant l'émission du message de l'affichage du cyberacteur dépend d'un certain nombre de facteurs, notamment de la taille du message contenant les données provenant du système de capteurs, et du temps que prennent les trois traitements ci-dessus —temps qui dépend également de la taille du message de données. L'évaluation de ces facteurs, associée à celle du nombre de messages de données envoyés par seconde par le système de capteurs, peut mener à l'adoption d'une architecture particulière.




```
transition t1 :
  triggered by : { {data, p2} }
  action :
     {  dataf = traitement1(data);
        matrices = traitement2(dataf);
        model = traitement3(matrices);
     }
```

Figure 5.5 Une première solution

Par exemple, les trois traitements pourraient être réalisés par un seul acteur, comme le montre la figure 5.5.

Sur cette figure [2], chaque fois que **Acteur1** est dans l'état **Running** et qu'il reçoit un message de type **{data}** sur sa porte p_2 qui lui provient du système de capteurs, il effectue séquentiellement les 3 traitements nécessaires. **dataf** contient les données provenant du système de capteurs après formatage; **matrices** contient les matrices remplies sur base des données formatées, et **model** contient le cyberacteur.

Il peut aussi être intéressant de faire faire les trois traitements par trois acteurs différents. Ils doivent bien sûr être réalisés séquentiellement pour un message donné, mais ces messages arrivent à un certain taux, et on peut par exemple imaginer que pendant qu'un acteur exécute le traitement 2 sur un message **m**, un acteur autre exécute déjà , en parallèle, le traitement 1 sur le message **m+1**, et ce afin d'augmenter la rapidité du flux de données. L'architecture en «pipeline» résultante est présentée à la figure 5.6.

[1] Cela dépend du type des données envoyées par les capteurs. Ces types sont repris dans l'annexe A1, page 97.
[2] Quelques libertés ont été prises par rapport au formalisme de ROOM.

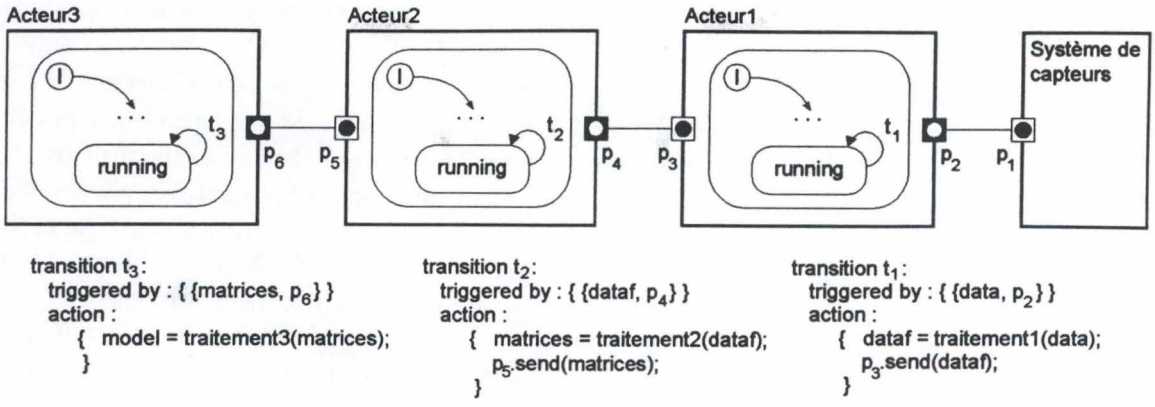


Figure 5.6 Une deuxième solution

Sur cette figure, chaque fois que Acteur1 reçoit un message de type {data} sur sa porte p2, il exécute le traitement 1 et envoie le résultat de ce traitement à Acteur2. Celui-ci exécute alors le traitement 2 et envoie son résultat à Acteur3 qui exécute le traitement 3. Pour pouvoir profiter pleinement de cette architecture, il faut que les 3 acteurs soient actifs le plus souvent possible : si Acteur3 traite le message m, Acteur2 doit déjà traiter le message m+1 et Acteur1 doit déjà traiter le message m+2.

Les solutions des figures 5.5 et 5.6 représentent des cas «extrêmes» : des solutions intermédiaires peuvent être envisagées, comme celles qui font l'objet de la figure 5.7.

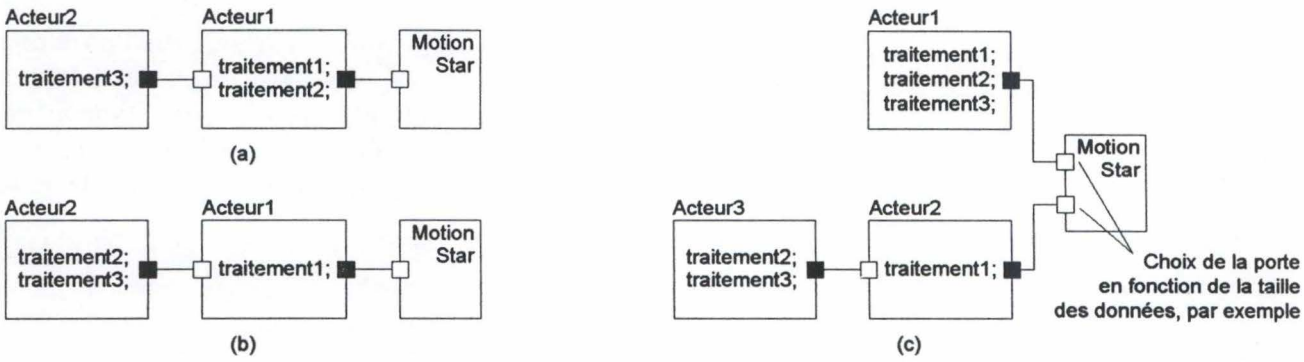


Figure 5.7 Des solutions intermédiaires

La solution optimale ne peut être trouvée qu'en évaluant les différents facteurs présentés plus haut : taille des données émises par le système de capteurs, temps que prennent les différents traitements à appliquer sur ces données, nombre de messages par seconde envoyés par ce système, etc.

La solution que nous avons choisie dans GabbyCapture est celle de la figure 5.7 (b), l'Acteur1 de cette figure étant aStreamActor dans GabbyCapture, et l'Acteur2 étant aMotionActor. Elle n'est la conséquence d'aucun choix précis : elle est plutôt le résultat qui a émergé progressivement d'une série de mini-décisions concernant les différentes parties de GabbyCapture. En effet, à aucun moment du développement de GabbyCapture (sauf peut-être à la fin) nous n'avons eu une vue d'ensemble des

différentes traitements qu'il fallait appliquer aux données provenant du système de capteurs.

Notre raisonnement se construisait étape par étape. Puisqu'il fallait récupérer les données en provenance des capteurs, nous avons d'abord écrit un ensemble de fonctions qui permettaient de récupérer ces données. Puisqu'un cyberacteur devait être chargé dans GabbyCapture, nous avons créé une structure de données permettant de l'accueillir. Nous ne nous sommes pas préoccupés, à ce stade-ci, du fait qu'elle était optimisée ou non pour être mise à jour en temps réel sur base des données provenant des capteurs, puisque nous n'avions pas encore réfléchi à ce problème. Ensuite, pour pouvoir appliquer les données sur le modèle, nous avons eu besoin de définir une structure contenant les matrices de données provenant des capteurs. Nous avons alors déterminé la manière de «lier» cette structure avec celle contenant le modèle —pour «attacher» les capteurs aux membres du personnage— puis nous avons écrit les mécanismes qui permettaient d'une part de mettre à jour cette structure sur base des données en provenance du système de capteurs et, d'autre part, de mettre à jour la structure contenant le modèle sur base de la structure contenant les données provenant du système de capteurs.

Tous ces choix qui ont été les nôtres durant ce processus ont toujours été effectués dans le cadre strict de la partie de code précise que nous étions en train de développer. Pour écrire un nouveau morceau de code, nous nous basions sur ceux que nous avions déjà écrits, comme s'ils constituaient des «données» au mini-problème que devait résoudre ce nouveau morceau. Bien sûr il nous arrivait de modifier ces «données» dans une certaine mesure, pour pouvoir faire «coller» l'ensemble.

Ce ne sont pas vraiment nos choix en eux-mêmes que nous critiquons ici, ni le fait qu'ils ont abouti à la création de morceaux de code optimisés ou non ; c'est plutôt la *portée* de ces choix. Aucun choix ne concernait l'*ensemble* du code, et la somme de morceaux de code optimisés séparément ne produit pas un ensemble de code optimisé.

Nous n'avons donc jamais réellement pris de décision en ce qui concerne la meilleure manière d'acheminer les données depuis le système de capteurs jusqu'au cyberacteur. Avec notre logique de «mini-choix», la question ne se posait même pas : la solution globale a émergé progressivement et nous avons espéré qu'elle serait suffisemment efficace. La question de savoir si notre solution était la meilleure ne s'est, à la limite, jamais posée. D'ailleurs, rien n'indique que notre solution soit la meilleure ou la plus mauvaise.

5.3.3. Parcours de l'arbre Model

• Lorsque **aMotionActor** reçoit de **anInterfaceActor** l'ordre de charger un nouveau cyberacteur dans sa structure **Model**, il effectue les actions suivantes (elles sont présentées de manière formelle à la page 114).

1□ Il exécute la méthode **Load** de l'objet **Model**, qui a pour effet de créer l'arbre et d'initialiser ses champs sur base des fichiers de GabbyCapture concernant un cyberacteur particulier. **Load** calcule ensuite le volume englobant local de chaque instance d'objet qui compose le cyberacteur (la définition d'un volume englobant est donné à la page 95).

2□ Il exécute la méthode **BoundingBox** de l'objet **Model**. Cette méthode parcourt l'arbre **Model**, lit les volumes englobants de chaque noeud, les transforme en coordonnées globales au cyberateur, et calcule le volume qui englobe tous ces volumes. Pour effectuer ces calculs, **BoundingBox** a besoin de calculer, pour chaque noeud, les matrices qui expriment la

position, l'orientation et le scaling de l'instance associée au noeud par rapport au *frame* global du cyberacteur.

3□ Il exécute la méthode **Display** de l'objet Model, qui consiste à effectuer un parcours de l'arbre, calculer pour chaque noeud les matrices qui expriment la position, l'orientation et le scaling de l'instance associée au noeud par rapport au *frame* global du cyberacteur, et enfin envoyer les messages nécessaires pour afficher l'instance d'objet associée à chaque noeud.

4□ Il exécute la méthode **AttachSensor** de l'objet Sensors pour associer le noeud racine de l'arbre avec le premier capteur.

BoundingBox et Display parcourent toutes les deux l'arbre Model et calculent toutes les deux les mêmes matrices pour chaque noeud, avant de réaliser, pour chaque noeud, un traitement différent. En n'effectuant qu'un seul parcours, dans lequel ces deux traitements auraient été effectués pour chaque noeud, nous aurions économisé un parcours d'arbre et le calcul des matrices pour chaque noeud. En fait, nous aurions même pu économiser un nouveau parcours en réalisant ces deux traitements directement lors de la création de l'arbre, dans la méthode Load.

Pourquoi ces parcours inutiles ? Quand nous avons écrit BoundingBox et Display, nous nous sommes dit que ces deux méthodes allaient devoir être utilisées par la suite, indépendamment de Load. Nous en avons donc fait deux méthodes séparées. Nous avions tort pour BoundingBox : elle est toujours exécutée après l'appel à la méthode Load, ce que nous ne savions pas au moment où nous l'avons écrite. Display est effectivement utilisée ailleurs, indépendamment des deux autres, ce qui peut donc justifier son existence, d'autant plus que réaliser un parcours d'arbre en trop n'est pas dramatique lors de l'affichage du cyberacteur.

• A chaque fois que la méthode Load de Model insère un nouveau noeud dans l'arbre, dans notre modèle ROOM elle envoie le nom du noeud («avant-bras gauche», «main droite», etc) à anInterfaceActor pour que celui-ci puisse l'afficher dans une de ses fenêtres (voir l'annexe A1, page 104). Dans le code de GabbyCapture, une méthode séparée existait indépendamment de Load et réalisait un parcours d'arbre supplémentaire pour envoyer les noms des noeuds à anInterfaceActor.

• A chaque fois que aMotionActor reçoit de aStreamActor un message de données formatées, comme cela est exposé à la page 109 de l'annexe A1, il effectue les opérations suivantes.

1□ Il exécute la méthode **UpdateDataMatrices** de l'objet Sensors, qui a pour effet de mettre à jour le contenu des matrices de Sensors sur base des données qui proviennent de aStreamActor (position et orientation des capteurs).

2□ Il exécute la méthode **UpdateLocalTransfos** de l'objet Model, qui a pour premier effet de mettre à jour les matrices de rotation et de translation du noeud racine de l'arbre sur base des matrices de translation et de rotation du premier capteur de l'objet Sensors. UpdateLocalTransfos parcourt ensuite l'arbre Model pour mettre à jour les matrices de rotation des noeuds auxquels sont associés des capteurs, sur base des données de l'objet Sensors.

3□ Il exécute la méthode **Display** de l'objet Model, pour afficher le cyberacteur dans une des fenêtres de anInterfaceActor.

UpdateLocalTransfos, pour mettre à jour la matrice de rotation locale à un noeud N, a besoin de

calculer la matrice d'accumulation en rotation de son noeud père. Elle calcule en fait la matrice d'accumulation en rotation, translation et scaling de ce noeud père, ce qui n'est pas gênant puisque la matrice de scaling de ce noeud est toujours égale à la matrice identité et que la matrice de translation n'influe pas sur les éléments de la matrice qui sont nécessaires pour pouvoir calculer la matrice de rotation locale du noeud N. Or c'est justement sur base de cette même matrice du noeud père que Display se base pour calculer les matrices qui sont nécessaires pour afficher l'instance d'objet associée au noeud N.

Il est donc possible d'économiser un parcours d'arbre, ainsi que le calcul de la matrice d'accumulation en rotation, translation et scaling de chaque noeud, ce qui représente un gain assez important, d'autant plus que les méthodes UpdateLocalTransfos et Display sont exécutées 25 fois par seconde.

Ceci dit, il est vrai que la méthode UpdateLocalTransfos ne parcourt pas *tous* les noeuds de l'arbre: après avoir mis à jour un noeud auquel est associé un capteur, elle teste si le sous-arbre dont ce noeud est la racine contient des noeuds auxquels des capteurs sont associés. Si ce n'est pas le cas, il ne parcourra pas ce sous-arbre. Pour réaliser ce test, chaque noeud de l'arbre dispose d'un booléen qui indique si des capteurs sont associés aux noeuds du sous-arbre dont il est la racine. UpdateLocalTransfos est donc optimisé pour ne parcourir que la portion de l'arbre qui doit être mise à jour [1].

UpdateLocalTransfos reflète très clairement la manière dont nous avons écrit GabbyCapture. Quand nous avons réfléchi à la manière de mettre à jour le cyberacteur, nous avions déjà écrit la méthode Display et nous avons considéré cette méthode comme un «donnée». Nous avons alors écrit UpdateLocalTransfos, tout en sachant qu'en ce qui concernait le réaffichage du cyberacteur nous allions utiliser Display.

UpdateLocalTransfos a été optimisée pour ne parcourir que les portions de l'arbre qui étaient modifiées par les données en provenance des capteurs. On peut considérer que les choix que nous avons effectués dans le cadre de cette méthode ont été pertinents. Cependant, si nous avions considéré le problème avec une vue plus globale, nous nous serions rendus compte que Display effectuait déjà dans une grande mesure ce que UpdateLocalTransfos faisait... et qu'écrire ces deux méthodes revenait à réaliser beaucoup de choses en double. Alors que nous croyions avoir optimisé le processus de mise à jour, nous avons en fait alourdi le processus de mise à jour *et* d'affichage.

5.3.4. Divers éléments de l'architecture de GabbyCapture

• Comme le montre la figure 5.8, l'arbre Model contient un noeud supplémentaire (le noeud de nom «root») par rapport aux noeuds qui définissent le cyberacteur dans les fichiers Gabby correspondants. Aucune instance d'objet n'est associée à ce noeud ; ses matrices de rotation, de translation et de scaling sont initialisées à la matrice identité. Pour autant qu'on ne modifie pas sa configuration initiale, ce noeud n'a donc aucun effet sur l'ensemble du cyberacteur : il a été placé là uniquement pour faciliter l'initialisation et la destruction de l'objet Model.

Cependant, comme nous le disions dans le chapitre 4, le capteur mesurant la position et l'orientation «globales» de l'animateur humain doit être associé avec le noeud racine du cyberacteur.

[1] Cette optimisation ne produit pas beaucoup d'effet si le cyberacteur est animé des pieds à la tête, puisque tous les noeuds (ou du moins une grande partie, étalés jusqu'aux feuilles de l'arbre) sont associés à des capteurs.

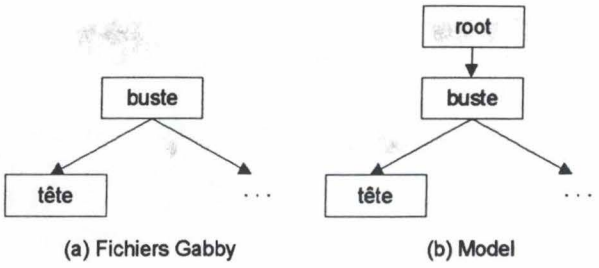


Figure 5.8 Modélisation du cyberacteur

Dans GabbyCapture, ce capteur est associé au noeud «root» (page 104 dans l'annexe A1), et non à la «véritable» racine de l'arbre, comme le 'buste' sur l'exemple de la figure 5.8.

Cette modification n'altère pas l'animation du personnage, puisque l'orientation et la position du noeud «root» sont répercutées sur la véritable racine de l'arbre et sur tous les autres noeuds. S'il était vraiment nécessaire de conserver ce noeud «root», cependant, nous aurions dû associer le premier capteur à la véritable racine du cyberacteur, et ce afin de ne pas compliquer inutilement la structure Model.

• Dans le code de GabbyCapture, la méthode AttachSensors de l'objet Sensors (définie dans l'annexe A1 à la page 107) a pour but d'attacher un capteur donné de cet objet à un certain noeud de l'objet Model. Pour ce faire, elle réalise d'abord des modifications au sein de Sensors, puis va modifier directement certains champs de Model. Dans notre modèle ROOM, une méthode AttachThisSensor a été définie pour l'objet Model (elle est présentée également à la page 107), et la méthode AttachSensors de Sensors appelle la méthode AttachThisSensor de Model plutôt que d'accéder directement aux champs de celui-ci.

Comme nous le disions dans le chapitre 3, il est possible avec C++ de définir des objets mais aussi de les court-circuiter, car l'accès direct aux champs des objets n'est pas interdit (du moins si on déclare ces champs avec la clause 'public'). Il est vrai que, pour certaines parties critiques du code, il peut être nécessaire, en terme de temps, de réaliser un accès direct au champ d'un objet plutôt que de passer par une méthode (ce n'était pas nécessaire dans l'exemple ci-dessus). Puisque la structure du code s'en trouve dégradée, il vaut mieux ne le faire que quand c'est indispensable. Cependant, sans avoir de vue d'ensemble du code, il est impossible de déterminer les endroits où des accès directs peuvent se justifier pour des raisons de performance.

• Chaque noeud de l'objet Model contient 4 éléments «logiques» : un pointeur vers son premier noeud fils, un pointeur vers son noeud père, un pointeur vers son premier noeud frère et un objet contenant les informations qui sont associées au noeud (nom du noeud, matrices de rotation de l'instance d'objet, référence vers l'objet-symbole, etc.). Les 3 premiers éléments concernent la structure de l'arbre Model et apparaissent comme autant de variables dans chaque noeud de l'objet Model. Le dernier élément logique ne correspond pas à une seule variable : il existe bien une variable pclNodeObject définie dans le noeud, mais elle ne contient qu'une partie des champs qu'elle aurait dû contenir. L'autre partie se retrouve sous la forme de variables définies directement dans le noeud. On trouvera, page 100, la définition C++ de la classe CMODEL dont Model est une instance.

Ce manque de structuration dans les noeuds de Model témoigne d'ajouts successifs qui ont été réalisés en fonction des besoins qui sont apparus progressivement lors de l'écriture de GabbyCapture. Ces ajouts ont entraîné la duplication des champs nécessaires pour stocker le volume englobant de

l'instance d'objet associée au noeud : ils apparaissent à la fois dans le noeud et dans son objet pclNodeObject. Une seule copie de ces champs a subsisté dans le modèle ROOM.

• Dans le code de GabbyCapture, un objet Cyberactor avait été défini : il contenait un objet Model et un ensemble de champs qui, logiquement, devaient faire partie de l'acteur anInterfaceActor (et qui n'avaient aucun rapport avec Cyberactor). Dans le modèle ROOM, Cyberactor a disparu, il ne reste plus que Model et les autres champs ont réintégré anInterfaceActor.

Ces exemples, et bien d'autres, n'affectent pas la directement la performance du logiciel. Pris de manière isolée, ils peuvent paraître anodins, mais constituent néanmoins autant de «défauts de structuration» avec lesquels il faudra compter lorsqu'il sera nécessaire de faire évoluer le code. Les décisions qui seront prises lors de ces évolutions se baseront sur ces lacunes et manqueront de pertinence, ou bien impliqueront la réécriture de certaines parties du logiciel afin de «corriger» ces défauts. Bien entendu, ces réécritures sont autant de pertes de temps qu'il faut si possible éviter.

A terme, les évolutions successives du logiciel produisent donc un effet «boule de neige» à partir des défauts de base. L'accumulation de ceux-ci rend le code difficilement compréhensible, non structuré mais aussi moins performant.

Conclusion

« Comment faire adopter à un personnage virtuel les mouvements d'un être humain ? »

Nous allons maintenant faire la synthèse des réponses qui, tout au long de ce travail, ont été apportées à cette question. Nous verrons qu'elles ont suscité d'autres interrogations, qui sont autant de pistes qui ne seront pas explorées dans le cadre de ce travail.

Dans l'introduction, nous avons vu que notre question initiale était sujette à deux interprétations différentes. Dans un premier sens, elle pouvait se reformuler : « Quels ont été les objectifs du projet GabbyCapture et quelles ont été les réalisations auxquelles il a donné lieu ? » ; et, dans un second sens: « Quels avantages peuvent apporter une démarche de développement *top-down* et l'utilisation d'un outil d'analyse spécifique pour l'écriture d'un logiciel comme GabbyCapture ? »

Les trois premiers chapitres ont tenté de répondre à la première des deux questions. Ces chapitres ont présenté le projet GabbyCapture et le contexte dans lequel il s'est déroulé ; ils ne donnent pas réellement matière à conclure.

Les chapitres 4 et 5 ont tenté de répondre à la seconde formulation de la question : « Quels avantages peuvent apporter une démarche de développement *top-down* et l'utilisation d'un outil d'analyse spécifique pour l'écriture d'un logiciel comme GabbyCapture ? »

Pour pouvoir y répondre, une telle démarche de développement a été entamée après l'écriture de GabbyCapture : une spécification des fonctionnalités de celui-ci a été écrite dans le chapitre 4. Elle concernait le «quoi» sans considérer la manière dont la solution informatique allait prendre forme — le «comment».

Le chapitre 5 s'est ensuite attaqué au «comment» et contient la description d'une architecture de GabbyCapture, écrite avec le langage ROOM (*Real-Time Object Oriented Modeling*). Cette architecture ne correspond cependant pas à celle qui aurait été écrite si on avait continué à appliquer la méthode

top-down entamée au chapitre 4. Au contraire, elle a été conçue dans le but initial de représenter le plus fidèlement possible le code qui avait été écrit durant la réalisation du projet GabbyCapture [1].

C'est l'écriture puis l'évaluation de cette architecture ROOM qui ont permis de mettre en évidence les avantages que pouvaient apporter une démarche de développement *top-down* et l'utilisation d'un outil d'analyse spécifique.

Cette architecture, qui représente donc dans une large mesure le code de GabbyCapture, nous a permis d'avoir une vue à la fois globale et précise de ce code, et ce pour deux raisons :

- parce qu'elle exprimait l'entièreté du code de GabbyCapture avec des concepts intégrés, simples, de haut niveau, masquant les détails d'implémentation ;
- parce que ces concepts étaient suffisemment formels pour être exécutables : aucune précision n'était enlevée par rapport au code.

Cette vision d'ensemble, associée au fait d'avoir réalisé la spécification de GabbyCapture, a permis de mettre en évidence certaines caractéristiques de la méthode de développement que nous avions utilisée pour écrire le code de GabbyCapture.

Tout d'abord, nous nous sommes rendus compte que la logique de développement que nous avions adoptée était une logique de «mini-choix» : leur portée était toujours réduite à la partie de code précise qui était en train d'être développée. Pour écrire un nouveau morceau de code, nous nous basions sur ceux que nous avions déjà écrits, comme s'ils constituaient des «données» au mini-problème que devait résoudre ce nouveau morceau. Aucun choix ne concernait l'ensemble du logiciel, aucun choix ne concernait la distinction de modules ou d'objets ayant des fonctionnalités précises. La logique de développement globale, telle quelle, n'existait pas : elle émanait plutôt d'un ensemble de mini-choix qui avait été effectués de manière cloisonnée.

Ensuite, nous avons réalisé que ces mini-choix ont engendré un code souffrant de 2 lacunes.

La première concerne la *performance* du logiciel. Les arguments que nous avons développés dans le chapitre 5 le montrent.

Par exemple, lors du réaffichage du personnage virtuel alors qu'il est animé en temps réel sur base de messages de données arrivant 25 fois par seconde du système de capteurs, deux méthodes sont appelées, parcourent toutes les deux l'arbre représentant le personnage et effectuent des calculs identiques. Un seul parcours aurait donc suffi. Mais quand nous avons réfléchi à la manière de réafficher le cyberacteur, nous avions déjà écrit une des deux méthodes et nous avons considéré cette méthode comme un «donnée» pour l'écriture de la seconde. Nous ne les avons pas envisagées dans une logique plus globale, et cela a affecté la performance du logiciel.

Autre exemple : nous avons écrit les différents traitements qu'il fallait appliquer aux données provenant du système de capteurs sans jamais (sauf peut-être à la fin du développement) avoir eu une vue d'ensemble de ces différents traitements. Nous n'avons donc jamais réellement pris de décision en ce qui concerne la meilleure manière d'acheminer les données depuis le système de capteurs jusqu'au personnage virtuel. Avec notre logique de «mini-choix», la question ne se posait même pas : la solution globale a émergé progressivement et nous avons espéré qu'elle serait suffisemment efficace. La question de savoir si notre solution était la meilleure ne s'est, à la limite, jamais posée, et rien n'indique qu'elle soit la meilleure ou la plus mauvaise.

La deuxième lacune concerne la *facilité de maintenance* du logiciel. Dans le chapitre 5, nous avons

[1] Quelques parties ont quand même été écrites sans plus tenir compte de ce code.

vu que l'existence d'un acteur de notre modèle ROOM était injustifiée ; divers autres éléments constituant autant de «défauts de structuration» ont également été mis en évidence. Les modifications successives qui seront réalisées lors des phases ultérieures de maintenance du logiciel se baseront sur ces défauts, risquent de les accentuer et de produire, à terme, un code non structuré, difficile à comprendre et difficile, voire impossible, à faire évoluer à nouveau.

Les avantages de l'adoption d'une démarche de développement *top-down* et de l'utilisation d'un outil d'analyse spécifique sont donc, d'une part, de pouvoir définir précisément le problème à résoudre et d'autre part, de pouvoir appréhender l'ensemble de l'architecture du logiciel. Nous avons montré que ces deux éléments étaient nécessaires pour pouvoir améliorer la qualité des choix et conserver ainsi toutes les chances d'obtenir une solution performante et maintenable.

Ceci dit, les outils d'analyse dans le domaine du temps réel, comme ROOM, sont encore peu répandus. Cela explique qu'aucune méthode d'analyse, à l'heure actuelle, n'ait encore satisfait Neurones. ROOM pourrait peut-être constituer un outil adapté à ses besoins futurs.

On pourrait objecter que l'adoption d'une méthode d'analyse ne se justifie pas pour des projets de petite envergure, qu'elle prend plus de temps que celle consistant à écrire du code directement ou encore qu'elle coûte plus cher.

Si on considère que la réalisation du logiciel GabbyCapture est un projet de petite envergure — il représente 5 mois/homme— alors l'adoption d'une méthode d'analyse peut donc se justifier même pour les petits projets.

Concernant le temps de développement, il est raccourci à long terme. En effet, au fur et à mesure de l'évolution du code, les tares du logiciel en termes de performance et de structure ont un effet «boule de neige». Il arrive un jour où la performance du logiciel devient inacceptable, et/ou le code devient très difficile à comprendre : il est alors indispensable de réécrire le logiciel, ce qui représente, bien sûr, une perte de temps énorme.

Le temps de développement à court terme n'est pas nécessairement plus long : le temps que l'on «perd» en écrivant la spécification et l'architecture peut être contrebalancé par le temps que l'on gagne à écrire le code nécessaire et suffisant. En effet, quand on développe avec une logique de mini-choix, on se rend par la suite compte que certaines parties de code étaient inadaptées —auquel cas il faut les réécrire pour les faire «coller» à l'ensemble— ou même inutiles.

Le coût entraîné par l'achat des outils de développement et par la formation du personnel peut être contrebalancé par des coûts de maintenance réduits et les bénéfices entraînés par la qualité accrue du logiciel.

Nous avons ainsi répondu à la deuxième question de ce travail. Ces réponses ont cependant suscité d'autres questions. Que fait-on si, par exemple, on ne connaît que certains éléments de la spécification du logiciel [1] ? Que fait-on s'il est nécessaire de fournir un prototype du logiciel le plus rapidement possible ? Une réponse, que nous ne développerons pas, serait d'utiliser une approche de développement «incrémentale», consistant à réaliser, pour une première partie du logiciel, sa spécification, son architecture puis son code, de recommencer ensuite avec une autre partie, et ainsi de suite. De cette manière, on «maximise» en quelque sorte la portée des choix que l'on peut faire dans les limites de nos contraintes.

[1] En d'autres termes : quelle méthode adopter pour réaliser les phases de maintenance du logiciel ?

Une autre question importante restera également sans réponse dans le cadre de ce travail : quelle méthode adopter pour développer l'architecture avec l'outil d'analyse choisi ? Pour réaliser l'architecture ROOM de GabbyCapture, nous nous sommes basés sur son code : ce n'est bien sûr pas une méthode conseillée ! Nous avons donc présenté un langage, ROOM, mais nous n'avons pas abordé la méthode qui aurait permis de l'utiliser pour produire une architecture en se basant uniquement sur la spécification.

Notre approche a été de représenter, en ROOM, le code que nous avions écrit dans le projet. C'est la vue d'ensemble qui en a découlé, associée au fait d'avoir écrit une spécification de GabbyCapture, qui nous a permis de tirer nos conclusions. Nous avons cependant exploré une partie de cette démarche à rebours. Des idées d'améliorations sont nées, mais la méthode de développement complète n'a pas été réalisée. Nos conclusions présagent néanmoins des bénéfices qui pourraient être engendrés par l'adoption d'une telle méthode.

Bibliographie

Les références qui suivent constituent une bibliographie «classique». Une bibliographie de sites Web est présentée à sa suite ; les références de ces sites se différencient des autres par une astérisque.

[Buc95]
G. Bucci, M. Campanai, P. Nesi, *Tools for specifying real-time systems*, Kluwer Academic Publishers, Boston, 1995.

[Coc97]
Donna Coco, «Motion Capture Advances», *Computer Graphics World*, novembre 1997, pages 37-44.

[Col98]
Florence Collard, «Doctoon, le 'pote' virtuel», *Le Quinzième Jour*, 28 mai-11 juin 1998, p.6.

[deK95]
de Keyser, V., Cours de psychologie cognitive, *Notes à l'usage des étudiants*, par J. Sougné.

[Edw97]
E. Angel, *Interactive computer graphics : a top-down approach with OpenGL*, Addison-Wesley, Reading, 1997.

[Fle68]
P. Fleury, J.-P. Mathieu, *Images optiques*, Eyrolles, Paris, 1968.

[Fol90]
J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer graphics : principles and practice*, Addison-Wesley, The Systems Programming Series, Reading, 1990.

[Ima]
J. Image, *Le dessin animé*, ?

[Nei93]
J. Neider, T. Davis, M. Woo, *OpenGL Programming Guide : the official guide to learning OpenGL, release 1*, Addison-Wesley, OpenGL Architecture Review Board, Reading, 1993.

[Rob97]
Barbara Robertson, « Pioneers and Performers », *Computer Graphics World*, novembre 1997, pages 46-52.

[Sel94]
B. Selic, G. Gullekson, P. T. Ward, *Real-time Object-Oriented Modeling*, Wiley, New York, 1994.

[Str91]
B. Stroustrup, *The C++ programming language*, Addison-Wesley, Reading, 1991.

Bibliographie WEB

[ObjecTime]* www.objectime.com

Site de la société ObjecTime, qui commercialise l'outil CASE ObjecTime basé sur le langage ROOM.

[Ascension]* www.ascension-tech.com

Site de la société Ascension Technologies, inc, qui commercialise le système de capteurs MotionStar.

[OpenGL]* www.sgi.com/Technology/OpenGL/

Site de la société Silicon Graphics, page d'introduction à OpenGL.

[Kaydara]* www.kaydara.com

Site de la société Kaydara, qui commercialise le logiciel FiLMBOX.

Annexe A1

Architecture ROOM de GabbyCapture

Nous allons présenter l'architecture ROOM de GabbyCapture en 7 étapes.

On commencera par l'exposé de la classe d'acteur MotionCaptureSystem (A1.1), dont l'unique instance est le système de capture de mouvements entier, comprenant GabbyCapture mais aussi le système de capteurs MotionStar. Nous avons décidé d'inclure MotionStar dans notre architecture afin d'obtenir une représentation homogène de l'ensemble de la solution technique répondant à notre problème de capture de mouvements.

Nous passerons ensuite en revue toutes les classes de protocoles utilisées dans cette architecture (A1.2) et enchaînerons avec l'exposé des autres classes d'acteurs du modèle : MotionActor (A1.3), InterfaceActor (A1.4), BirdsActor (A1.5), StreamActor (A1.6) et DispatchActor (A1.7).

Dans notre modèle, il n'existe qu'une seule instance de chacune de ces 5 classes. Les instances des classes MotionActor, InterfaceActor, StreamActor et DispatchActor constituent la représentation ROOM de GabbyCapture. L'instance de BirdsActor représente le système MotionStar.

La classe d'acteur InterfaceActor est d'un type particulier : elle dispose d'une représentation visuelle (sa seule instance représente l'interface utilisateur de GabbyCapture). Ce type d'acteur n'existe pas dans ROOM : nous l'avons imaginé afin de ne pas devoir utiliser un second formalisme pour pouvoir modéliser l'interface utilisateur de GabbyCapture. Nous pourrons donner une représentation ROOM de la structure de cette classe InterfaceActor mais nous ne pourrons donner qu'une description informelle de son comportement. Bien entendu, nous présenterons également sa représentation visuelle, sous forme de fenêtres [1].

[1] Le plus logique aurait été de présenter les fenêtres de GabbyCapture, écrites en X Window. Cependant, aucune machine n'étant disponible chez Neurones pour exécuter GabbyCapture et réaliser les copies d'écran nécessaires au moment où j'ai voulu le faire, j'ai donc redessiné les fenêtres à l'aide de Microsoft Developer Studio et Microsoft Visual C++ 4.2. Ce sont des copies de ces nouvelles fenêtres, sémantiquement équivalentes à celles de GabbyCapture, que nous retrouverons dans ce travail.

En ce qui concerne la classe BirdsActor, puisqu'elle décrit un composant du système que nous n'avons pas réalisé, nous décrirons uniquement en ROOM sa structure générale et nous ne donnerons qu'une description informelle de son comportement.

Pour les classes d'acteur MotionActor, StreamActor et DispatchActor, nous présenterons en ROOM leur structure et leur comportement.

A1.1. La classe d'acteur 'MotionCaptureSystem'

La classe d'acteur principale de notre design ROOM est MotionCaptureSystem. Sa structure et son comportement sont exposés ci-dessous.

A1.1.1. Structure

La structure de MotionCaptureSystem fait l'objet de la figure A1.1.

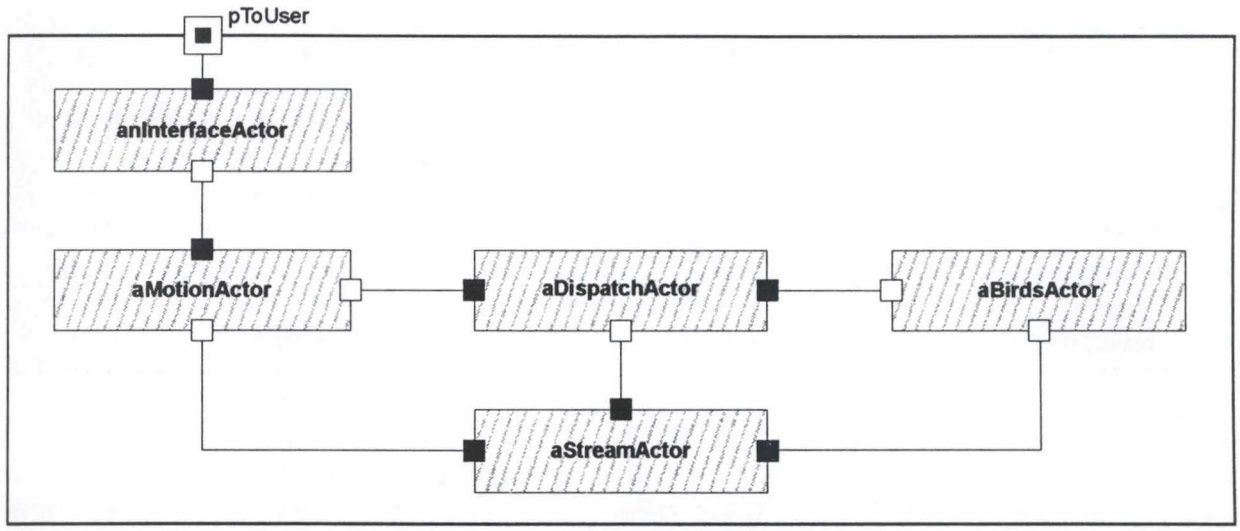


Figure A1.1 La classe d'acteur MotionCaptureSystem

Cette classe contient 5 instances d'acteurs, dont les classes seront exposées dans les points A1.3 à A1.7. Nous verrons également dans ces points à quelles classes de protocoles les portes de ces instances font référence (ces classes de protocoles font l'objet du point A1.2). Ces classes déterminent les messages qui pourront transiter sur les liaisons reliant les portes des 5 instances d'acteurs.

MotionCaptureSystem dispose également d'une porte-relais qui permet de relayer vers l'utilisateur du système (vers anInterfaceActor) les messages qui proviennent de anInterfaceActor (de l'utilisateur) et qui lui sont destinés.

A1.1.2. Comportement

MotionCaptureSystem n'a pas de comportement propre : son comportement est entièrement

déterminé par celui de ses 5 acteurs contenus. Nous renvoyons donc le lecteur aux points A1.3 à A1.7 dans lesquels on aborde les comportements des classes d'acteurs dont les 5 acteurs contenus sont des instances.

A1.2. Classes de protocoles

La figure A1.2 présente les classes de protocoles qui sont utilisées dans notre design. Le protocole permettant à anInterfaceActor de communiquer avec l'utilisateur du système n'y est pas présent : ce sera le seul qui ne sera pas abordé de manière formelle.

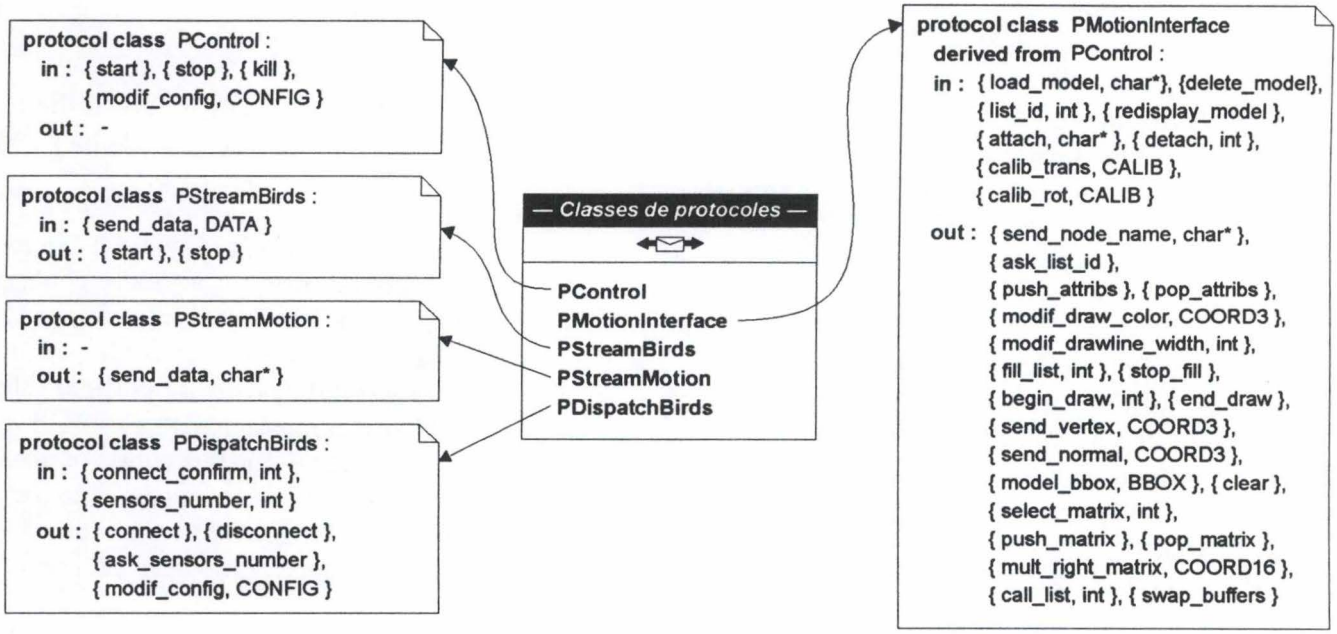


Figure A1.2 Classes de protocoles

Les commentaires qui suivent sont organisés comme ceci : pour chaque classe de protocole,

- chaque type de message (' *in* ' et ' *out* ') est commenté ;
- les acteurs qui prennent part aux communications point-à-point déterminées par la classe, c'est-à-dire les acteurs qui disposent d'au moins une porte qui est du type de la classe et qui est liée à celle d'un autre acteur, sont indiqués en caractères gras.

Le type des variables qui sont utilisées dans les messages des classes de protocoles sont des types C++, prédéfinis ou non. Les définitions des constantes et des types non prédéfinis qui sont utilisés dans ces messages sont données à la fin de cette section, au point A1.2.6.

A1.2.1. PControl

In

{start}, {stop}, {kill} et {modif_config, CONFIG} sont envoyés par **anInterfaceActor** pour signifier

à **aMotionActor** de, respectivement, lancer l'animation du modèle, la stopper, clore le système et modifier la configuration du système (en lui donnant la nouvelle configuration sous la forme d'une variable de type CONFIG). Ces 4 messages sont également utilisés par aMotionActor vers **aDispatchActor**, et par ce dernier vers **aStreamActor** dans le même but.

A1.2.2. PMotionInterface

PMotionInterface est une classe de protocole fille de PControl : elle hérite donc des 4 messages de PControl. Nous ne reviendrons par sur ceux-ci. PMotionInterface est utilisé uniquement entre **aMotionActor** et **anInterfaceActor**. Si on se réfère à la conjugaison des portes qui relient ces deux acteurs sur la figure A1.1, on déduit que les messages ' *in* ' du protocole vont de anInterfaceActor vers aMotionActor.

In

{load_model, **char***} est utilisé par anInterfaceActor pour demander à aMotionActor d'initialiser Model. La variable de type char* qui accompagne le message contient le nom du fichier texte principal présent dans le catalogue.

{delete_model} enjoint aMotionActor de réinitialiser Model avec un modèle vide.

{list_id, **int**} est envoyé à aMotionActor dès que celui-ci demande, par un {ask_list_id} (voir les messages ' *out* ' ci-dessous), un identifiant de *display list* OpenGL non utilisée. Cet échange de messages est implémenté par l'appel à la fonction OpenGL glGenLists. Nous verrons au point A1.4.2 que toutes les structures OpenGL, dans notre design, sont considérées comme faisant partie des structures de données relatives à la fenêtre openGLWin de anInterfaceActor.

{redisplay_model} demande à aMotionActor d'envoyer à nouveau les messages nécessaires (voir messages ' *out* ' ci-dessous) pour afficher le modèle. anInterfaceActor fait cette requête à chaque fois que openGLWin est cachée par une autre fenêtre puis de nouveau affichée.

{attach, **char***} enjoint aMotionActor d'associer le capteur dont l'identifiant correspond au char[0] de la donnée associée au message, avec le noeud de Model dont le nom est stocké à partir de char[1].

{detach, **int**} enjoint aMotionActor de détacher le capteur dont l'identifiant est donné par int.

{calib_trans, CALIB} et {calib_rot, CALIB} permettent à anInterfaceActor d'envoyer à aMotionActor les matrices de calibration, respectivement en translation et rotation, associées à un capteur donné via une variable de type CALIB.

Out

{send_node_name, **char***} est utilisé par aMotionActor pour envoyer à anInterfaceActor le nom d'un noeud de Model (char*).

{ask_list_id} permet à aMotionActor de demander un identifiant de *display list* non encore utilisée.

{push_attribs} est utilisé par aMotionActor pour demander à anInterfaceActor de créer un nouveau jeu d'attributs —strictement identique à celui qui occupait le sommet de la pile des attributs OpenGL— et de l'empiler sur cette pile. Le jeu d'attributs actif est toujours celui situé sur le sommet de la pile. L'envoi d'un message {push_attribs} est implémenté par l'appel à la fonction OpenGL glPushAttribs.

{pop_attribs} enjoint anInterfaceActor de dépiler le jeu d'attributs qui occupe le sommet de la pile d'attributs OpenGL. L'envoi d'un tel message est implémenté par glPopAttribs.

{modif_drawcolor, COORD3} et {modif_drawline_width, int} sont envoyés à anInterfaceActor pour modifier respectivement la couleur de dessin et l'épaisseur du trait de dessin (ce sont 2 attributs faisant partie du jeu d'attributs OpenGL). Les appels aux fonctions OpenGL

glLineWidth et glColor3x implémentent l'envoi de ces deux messages. La couleur de dessin est spécifiée par un code RGB stocké dans une variable de type COORD3.

{fill_list, int} indique à anInterfaceActor que les messages qui vont suivre celui-ci et qui sont relatifs au dessin (comme {begin_draw, int} ou {send_vertex, COORD3} ci-dessous) sont destinés à remplir la *display list* dont l'identifiant est donné par la variable de type int, et ce jusqu'à ce qu'un message {stop_fill} lui arrive. L'envoi de {fill_list, int} et de {stop_fill} sont implémentés par glNewList et glEndList.

{begin_draw, int} et {end_draw} sont envoyés par aMotionActor pour signifier à anInterfaceActor que tous les messages qui seront envoyés après l'envoi de {begin_draw, int} et avant l'envoi de {end_draw}, et qui sont de la forme {send_vertex, COORD3} (ci-dessous) spécifient des sommets faisant partie d'une forme géométrique dont le type est donné par la variable de type int, et qui peut prendre les valeurs prédéfinies GL_LINES, GL_LINE_LOOP, GL_QUADS et GL_TRIANGLES. {begin_draw, int} et {end_draw} sont implémentés par glBegin et glEnd.

{send_vertex, COORD3} est envoyé par aMotionActor pour spécifier un sommet (de type COORD3) faisant partie d'une forme géométrique (voir {begin_draw, int} ci-dessus). L'envoi de ce message est implémenté par glVertex3x.

{send_normal, COORD3} est envoyé par aMotionActor, juste avant d'envoyer un message de type {send_vertex, COORD3}, pour spécifier un vecteur (de type COORD3) relatif à ce sommet. Ce vecteur est nécessaire pour calculer, en fonction de la luminosité définie pour la scène dans laquelle prend place le modèle, la couleur de la surface dont le sommet est l'un des points. Dans le cadre de ce travail, nous n'entrerons pas dans le détail des questions concernant la luminosité. L'envoi de {send_normal, COORD3} est implémenté par glNormal3x.

{model_bbox, BBOX} est utilisé par aMotionActor pour envoyer à anInterfaceActor les coordonnées (dans une variable de type BBOX) déterminant le volume englobant du modèle, c'est-à-dire un parallélipipède rectangle dans lequel le modèle est inscrit. Ces coordonnées sont au nombre de 2 (BBOX contient deux variables de type COORD3) : elles correspondent aux points min et max de la figure A1.3.

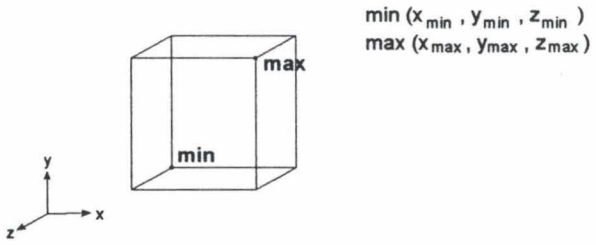


Figure A1.3 Le volume englobant de Model

{clear} enjoint anInterfaceActor de nettoyer la zone de dessin (de la « peindre » entièrement en blanc). Comme nous avons décidé de travailler en *double buffering*, {clear} nettoie en fait le *back buffer*.

{select_matrix, int} est envoyé par aMotionActor pour rendre courante la matrice OpenGL identifiée dans le message par la valeur de type int (constantes GL_MODELVIEW ou GL_PROJECTION). Les messages comme {push_matrix} ou {mult_right_matrix, COORD16} (ci-dessous) affectent la matrice courante. {select_matrix, int} est implémenté par glMatrixMode.

{push_matrix} est utilisé par aMotionActor pour demander à anInterfaceActor de créer une nouvelle matrice du type de la matrice courante, strictement identique à celle qui occupait le sommet de la pile des matrices du type de la matrice courante (Modelview ou Projection), et de l'empiler sur cette pile. La matrice courante est toujours celle située sur le sommet de la pile des matrices du type de la matrice courante . L'envoi d'un message {push_matrix} est implémenté par glPushMatrix.

{pop_matrix} enjoint anInterfaceActor de dépiler la matrice qui occupe le sommet de la pile des matrices du type de la matrice courante. L'envoi d'un tel message est implémenté par glPopMatrix.

{mult_right_matrix, COORD16} demande à anInterfaceActor de post-multiplier la matrice courante OpenGL par la matrice contenue dans la variable de type COORD16. L'envoi d'un tel message est implémenté par glMultMatrixd.

{call_list, **int**} demande à anInterfaceActor de dessiner sur le *back buffer* le contenu de la *display list* spécifiée.

{swap_buffers} demande à anInterfaceActor d'intervertir les deux *buffers*, ce qui rend le *back buffer* visible dans la fenêtre openGLWin.

A1.2.3. PStreamBirds

PStreamBirds est utilisé uniquement entre **aBirdsActor** et **aStreamActor**. Si on se réfère à la conjugaison des portes qui relient ces deux acteurs sur la figure A1.1, on déduit que les messages ' *in* ' du protocole vont de aBirdsActor vers aStreamActor.

In

{send_data, DATA} est utilisé par aBirdsActor pour envoyer à aStreamActor une variable de type DATA qui contient toutes les mesures (position et/ou orientation) relatives à l'ensemble des capteurs à un instant donné. Chaque capteur peut envoyer un format différent de données.

Out

Quand aBirdsActor reçoit un message {start}, il envoie des messages {send_data, DATA} au rythme de Nm par seconde, jusqu'à ce qu'il reçoive un message {stop}.

A1.2.4. PStreamMotion

Out

{send_data, **char***} est utilisé par **aStreamActor** pour envoyer à **aMotionActor** une variable de type **char*** contenant les données des capteurs formatées. La structure précise de cette variable est abordée sur la figure A1.21 (à droite sur la figure).

A1.2.5. PDispatchBirds

PDispatchBirds est utilisé uniquement entre **aBirdsActor** et **aDispatchActor**. Si on se réfère à la conjugaison des portes qui relient ces deux acteurs sur la figure A1.1, on déduit que les messages ' *in* ' du protocole vont de aBirdsActor vers aDispatchActor.

In

{connect_confirm, **int**} est envoyé par aBirdsActor en réponse à la réception d'un {connect} (ci-dessous). La variable de type int détermine la réussite de la tentative de connexion.

{sensors_number, **int**} est utilisé par aBirdsActor pour envoyer le nombre de capteurs qu'il contient et est envoyé en réponse à l'envoi, par aDispatchActor, d'un {ask_sensors_number} (ci-dessous).

Out

{connect} et {disconnect} sont utilisés par aDispatchActor pour se connecter et de se déconnecter à aBirdsActor.

{ask_sensors_number} est utilisé par aDispatchActor pour demander le nombre de capteurs de aBirdsActor.

{modif_config, CONFIG} enjoint à aDispatchActor de remplacer sa configuration avec celle qui lui est donnée sous la forme d'une variable de type CONFIG.

A1.2.6. Constantes et types de données

Les constantes et les types non prédéfinis qui ont été utilisés dans les classes de protocoles sont définis ci-après.

A1.2.6.1. Constantes

```
/* Les 6 constantes qui suivent ont la même signification que les constantes OpenGL du même
nom */

#define GL_LINES            0,
#define GL_LINE_LOOP        1,
#define GL_QUADS            2,
#define GL_TRIANGLES        3,

#define GL_MODELVIEW        0,
#define GL_PROJECTION       1,

/* Les constantes ci-dessous déterminent le format des données envoyées par un capteur.
Elles sont utilisées comme valeurs pour les champs 'dataFormats[x]' du type CONFIG (voir
ci-dessous).  */

#define FLOCK_NOBIRDDATA      = 0,        /* Aucune donnée */
#define FLOCK_POSITION        = 0x31,
        /* Position du capteur uniquement, sous forme de 3 valeurs représentant les distances
        (en pouces) auxquelles se situe le centre du frame du capteur sur les axes x, y et z
        du frame global de aBirdsActor. */
#define FLOCK_ANGLES          = 0x32,
        /* Orientation du capteur uniquement, sous forme de 3 valeurs d'angles (en degrés)
        déterminant l'orientation des axes du frame local au capteur par rapport à ceux du
        frame global de aBirdsActor. */
#define FLOCK_MATRIX          = 0x93,
        /* Orientation du capteur uniquement, sous forme d'une matrice de rotation 4x4
        représentant l'orientation du frame local du capteur par rapport à celui du frame global
        de aBirdsActor. */
#define FLOCK_POSITIONANGLES= 0x64,
        /* Position du capteur sous forme de 3 valeurs (les mêmes que celles de
        FLOCK_POSITION) et orientation du capteur sous forme de 3 angles (les mêmes que
        ceux de FLOCK_ANGLES). */
#define FLOCK_POSITIONMATRIX = 0xC5
        /* Position du capteur sous forme de 3 valeurs (les mêmes que celles de
```

```
FLOCK_POSITION) et orientation du capteur sous forme d'une matrice (la même que
celle de FLOCK_MATRIX). */

/* Les 3 constantes ci-dessous déterminent les formats possibles pour une valeur envoyée par
un capteur. En vertu des constantes ci-dessus, une mesure particulière de capteur peut être
de 3 formats différents : */

#define POSITION_DATA    = 0,    /* - une valeur de position (en pouces)      */
#define ANGLES_DATA      = 1,    /* - une valeur d'orientation (en degres)    */
#define MATRIX_DATA      = 2;    /* - une valeur d'orientation (cos ou sin)   */
```

A1.2.6.2. Types

```
typedef struct                   /* Configuration de aBirdsActor :            */
{ shortsensorsNb;                /*  - nombre de capteurs,                    */
  char sampleRate;               /*  - nombre de messages par seconde         */
                                 /*    envoyés par les capteurs               */
  char *dataFormats;             /*  - format des données de chaque capteur   */
} CONFIG;

typedef double COORD3[3];        /* 3 valeurs réelles pour spécifier un point, */
                                 /* un vecteur, des valeurs de calibration,    */
                                 /* un code RGB, etc.                          */

typedef struct                   /* Informations de calibration :             */
{ int          sensorId;         /*  - n° du capteur concerné,                */
  COORD3 calibData;              /*  - valeurs de calibration (rotation ou    */
                                 /*    translation) du capteur                */
} CALIB;

typedef COORD3 BBOX[2];          /* 2 sommets déterminant une bounding box    */
                                 /* (un parallélipipède englobant)            */

typedef double COORD16[16];      /* 16 valeurs réelles pour modéliser une     */
                                 /* matrice 4x4                               */

typedef struct
{ HEADER          head;
  unsigned char  *buffer;
} DATA;
/* Données contenant les mesures relatives à tous les capteurs, à un instant donné, envoyées
par aBirdsActor. La structure précise de head et de buffer est abordée au point A1.5.3. */
```

A1.3. La classe d'acteur 'MotionActor'

La figure A1.4 présente la structure et le comportement de la classe d'acteur **MotionActor**, dont l'unique instance **aMotionActor** constitue l'acteur central de notre modèle, celui qui contient les structures de données accueillant le modèle 3D et les données relatives aux capteurs.

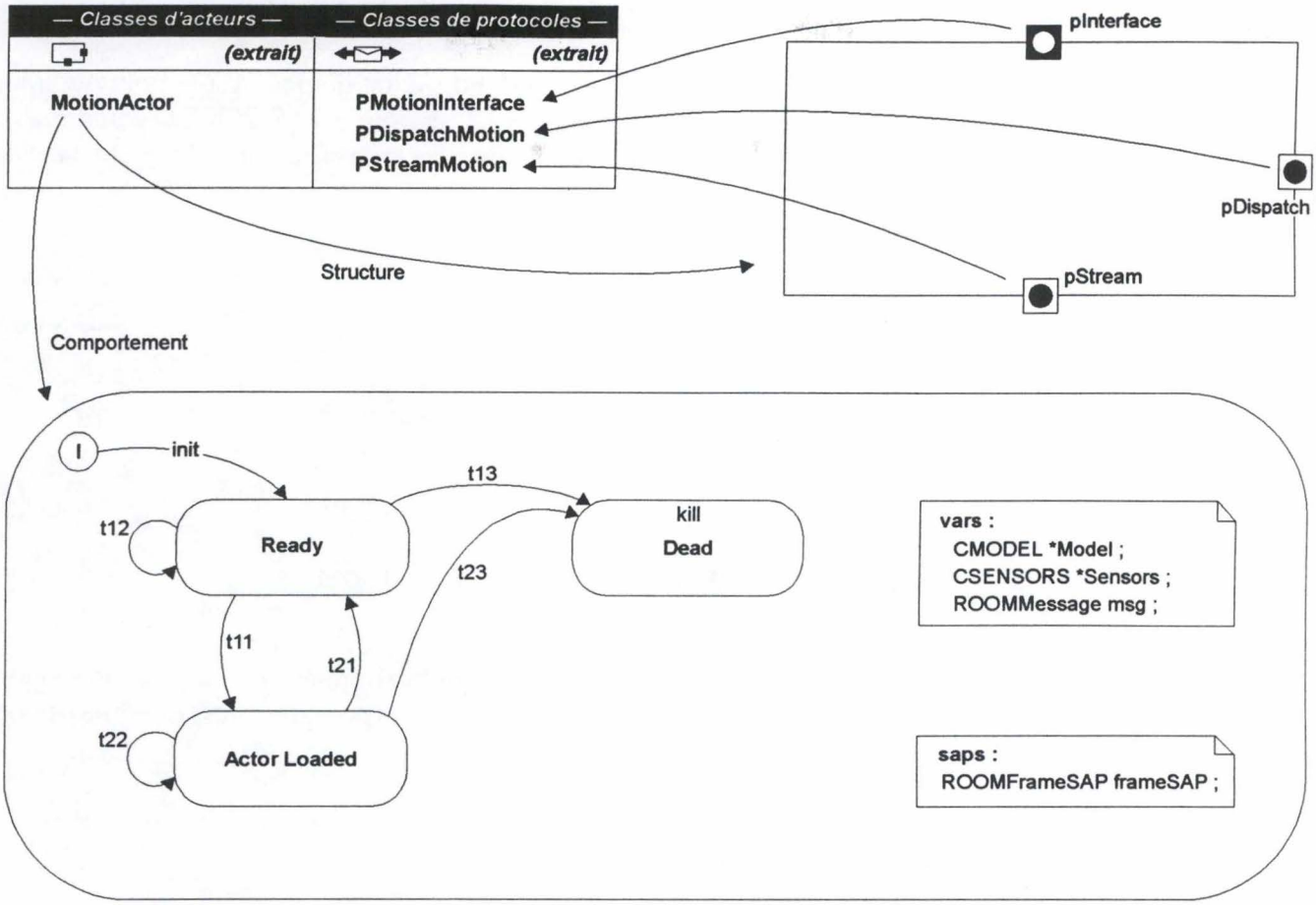


Figure A1.4 Structure et comportement de MotionActor

A1.3.1. Structure

Toute instance de MotionActor dispose de 3 portes lui permettant de communiquer selon les protocoles indiqués par les flèches. Dans notre modèle, les portes de la seule instance aMotionActor de la classe MotionActor sont reliées à anInterfaceActor (porte pInterface), aStreamActor (porte pStream) et aDispatchActor (porte pDispatch).

A1.3.2. Comportement

Dans cette section, nous allons expliciter tous les éléments qui apparaissent sur le ROOMChart (le diagramme états-transitions) de la figure A1.4. Nous commencerons par les variables et le SAP (A1.3.2.1): la variable Model —nous donnerons l'interface C++ de la classe CMODEL et de la classe COBJECT3D qui, comme nous le verrons, est utilisée par CMODEL—, la variable Sensors —nous définirons la classe CSENSORS—, la variable msg et enfin le SAP frameSAP. Nous commenterons ensuite les états du ROOMChart (A1.3.2.2) et terminerons (A1.3.2.3) par la définition des transitions du ROOMChart ainsi que par l'exposé des méthodes des classes CMODEL, COBJECT3D et CSENSORS. L'exposé de ces méthodes consistera à donner un commentaire textuel pour chacune d'entre elles. Ce commentaire sera parfois accompagné de l'algorithme de la méthode en version simplifiée.

A1.3.2.1. Variables et SAP

Model, les classes CMODEL et COBJECT3D □ aMotionActor dispose d'un pointeur Model pointant vers une instance de CMODEL, qui représente le modèle 3D. CMODEL constitue une implémentation d'arbre. La figure A1.5 présente cette implémentation; l'interface de la classe CMODEL est présentée ci-dessous.

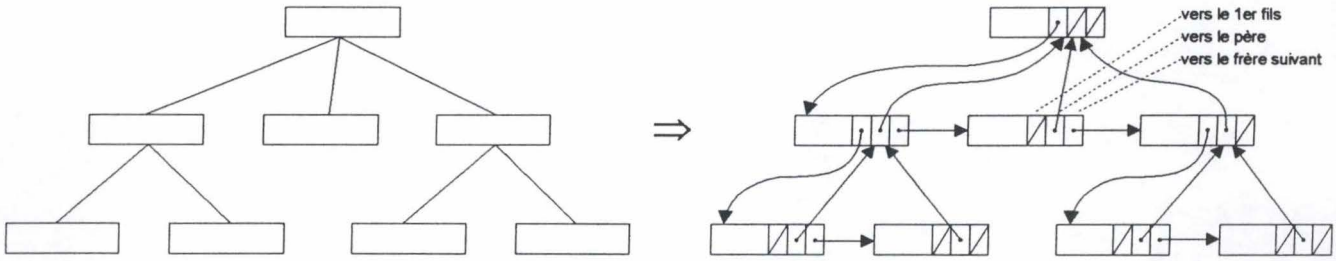


Figure A1.5 L'implémentation d'un arbre (CMODEL)

```
class CMODEL          // CMODEL définit un noeud de l'arbre (qu'on appellera pour la clarté 'noeud
1 {                   // courant'), les infos qui y sont associées et ses pointeurs vers d'autres noeuds.
2     public :
3         char            *szNodeModelName;
4         COBJECT3D       *pclNodeObject;
5         int             iDisplayListID;
6         TSENSORDATA     *pstSensorData;
7         int             iSensorDisplayListID;
8         BOOL            bSonsHaveSensors;
9         BOOL            bNextHaveSensors;
10        CMATRIX         clAccumRot;
11
12        CMODEL          *pclSons;
13        CMODEL          *pclFather;
14        CMODEL          *pclNext;
15
16        CMODEL();
17        ~CMODEL();
18        void Load(char *szModel);
19        CMODEL *AttachThisSensor(TSENSORDATA *sensorData, char *szNodeModelName);
20        void UpdateSensorsFlags();
21        void UpdateLocalTransfos();
22        void BoundingBox();
23        void Display();
24    private :
25        void _Display();
26 };
```

* Commençons par les lignes 12 à 14 : elles contiennent les déclarations des pointeurs du noeud courant
  - vers le premier de ses noeuds fils,
  - vers son noeud père,
  - vers son 'frère' qui le suit directement : tous les fils d'un noeud sont donc

liés entre eux.
* Les autres variables de CMODEL concernent les infos associées au noeud courant.
Revenons à la ligne 3 : szNodeModelName est le nom du noeud courant. Ligne 4: pclNodeObject pointe vers un objet de type COBJECT3D (ci-dessous), qui contient
- les matrices déterminant la position et l'orientation du *frame* local du noeud courant par rapport au *frame* de son noeud père,
- une variable déterminant le volume englobant de l'objet associé au noeud courant.

Ligne 5 : iDisplayListID contient l'identifiant de la *display list* dans laquelle l'objet associé au noeud courant est stocké. En effet, Model ne stocke pas la représentation graphique de ses objets : on verra que lors de l'initialisation de Model, les objets sont directement envoyés à anInterfaceActor pour être stockés dans les *display lists* OpenGL, qui sont des structures optimisées pour l'affichage d'informations graphiques.
Ligne 6 : pstSensorData pointe vers l'éventuel capteur (de l'objet Sensors) qui est attaché au noeud courant (la structure TSENSORDATA sera abordée quand nous parlerons de l'objet CSENSORS plus bas). Ligne 7 : iSensorDisplayListID est l'identifiant de la *display list* dans laquelle se trouve la représentation graphique (un cube dont on ne voit que les arêtes) du capteur attaché éventuellement au noeud courant.
Lignes 8 et 9 : booléens indiquant
- si des capteurs sont attachés à au moins un noeud du sous-arbre dont la racine est le noeud courant (bSonsHaveSensors) ;
- si des capteurs sont attachés à au moins un noeud appartenant aux sous-arbres dont les frères du noeud courant sont les racines (bNextHaveSensors).

Ces booléens sont utiles pour éviter de parcourir l'arbre inutilement dans certaines circonstances.
Ligne 10 : clAccumRot, matrice d'accumulation du noeud courant, détermine l'orientation du *frame* local de ce noeud par rapport au *frame* global du modèle 3D. Les instances de la classe CMATRIX sont des matrices 4x4 sur lesquelles une série d'opérations classiques peuvent être appliquées: multiplications, inversion, etc. Nous ne reviendrons pas sur cette classe.
* Les méthodes de l'objet (lignes 16 à 25) seront détaillées et commentées au point A1.3.2.3, lorsque nous parlerons des transitions du ROOMChart de aMotionActor.
* Aucun type booléen prédéfini n'existe en C ou en C++ ; BOOL est donc défini comme suit: **typedef** int BOOL;.
* Quelques critiques au sujet de CMODEL :
- on ne peut pas réellement justifier le choix de répartir les infos relatives à un noeud dans pclNodeObject (COBJECT3D) *et* dans le noeud lui-même (CMODEL). La solution la plus propre aurait été de placer toutes les informations relatives au noeud dans l'objet COBJECT3D (ou toutes dans CMODEL).
- Toutes les variables (lignes 3 à 16) auraient dû être déclarées 'private' et auraient dû n'être accessibles qu'au travers des méthodes de l'objet.
- Une seule *display list* aurait suffi pour stocker la représentation visuelle d'un capteur: or nous avons une *display list* par noeud ! De plus, l'identifiant de cette *display list* aurait dû faire partie de la classe CSENSORS, puisqu'il s'agit d'une information relative aux capteurs et non au modèle.

```
class COBJECT3D
1 {
2 public:
3     CMATRIX  clRot;
4     CMATRIX  clTrans;
5     COORD3  stBoundingBox[2];
6
7     COBJECT3D();
8     ~COBJECT3D();
9     void Object3DLoad(char *szFilename, int *iDisplayListID, int *iSensorDisplayListID);
10};
```

* clRot et clTrans (lignes 3 et 4) sont les matrices déterminant la position et l'orientation du *frame* local du noeud courant par rapport au *frame* de son noeud père.
* stBoundingBox (ligne 5) est la variable déterminant le volume englobant de l'objet associé au noeud courant.

* Les méthodes de l'objet (lignes 7 à 9) seront détaillées et commentées au point A1.3.2.3, lorsque nous parlerons des transitions du ROOMChart de aMotionActor.

Sensors et la classe CSENSORS □ aMotionActor dispose d'un pointeur Sensors pointant vers une instance de CSENSORS, qui représente la structure de données qui contient les informations relatives aux capteurs. L'interface de cette classe est présentée ci-dessous.

```
class CSENSORS
1 {
2     public :
3         int             iNbOfSensors;
4         char            sampleRate;
5         CMATRIX         clBirdToCaptureMatrix;
6         TSENSORDATA *pstSensors;
7
8         CSENSORS(CONFIG currentConfig);
9         ~CSENSORS();
10        void AttachSensor(int iSensorId, char *szNodeModelName, CMODEL *Model);
11        void DetachSensor(int iSensorId, CMODEL *Model);
12        void ModifConfig(CONFIG currentConfig);
13        void UpdateCalibrationTrans(CALIB calib);
14        void UpdateCalibrationRot(CALIB calib);
15        void UpdateDataMatrices(char* data);
16 };
```

* iNbOfSensors est le nombre de capteurs contenus dans aBirdsActor (ligne 3). sampleRate (ligne 4) est le nombre de messages de données envoyés par seconde par les capteurs.
* Comme le montre la figure A1.6, le *frame* global de la scène (dans laquelle se trouve le modèle 3D) et le *frame* global de aBirdsActor sont similaires dans le sens où ce sont tous les deux des *right-handed coordinate systems* (l'axe des z positifs sort de la feuille). Ils sont cependant orientés différemment. Chaque mesure de capteur doit donc subir une rotation pour pouvoir être exprimée en fonction du *frame* de la scène. En pratique, chaque mesure de capteur est multipliée par clBirdToCaptureMatrix (ligne 5).

Figure A1.6 Différence entre frames

* pstSensors, ligne 6, est un tableau d'éléments de type TSENSORDATA (ci-dessous) qui contient les infos relatives à un capteur. La longueur de ce tableau est donc égale au nombre de capteurs.
* Les méthodes de l'objet (lignes 8 à 15) seront détaillées et commentées au point A1.3.2.3, lorsque nous parlerons des transitions du ROOMChart de aMotionActor.
* Une critique au sujet de CSENSORS: toutes les variables auraient dû être déclarées 'private' et auraient dû n'être accessibles qu'au travers des méthodes de l'objet.

```
1 typedef struct {
2         CMATRIX  clDataTrans;
3         CMATRIX  clDataRot;
4
5         COORD3   stCentCalibTrans;
6         COORD3   stCentCalibRot;
7         CMATRIX  clCentCalibTrans;
8         CMATRIX  clCentCalibRot;
9
10        CMODEL   *pclAttachedModel;
11        char     dataFormat;
12 } TSENSORDATA;
```

* clDataTrans et clDataRot (lignes 2 et 3) sont les matrices déterminant respectivement la position et l'orientation du capteur, exprimées par rapport au *frame* global de aBirdsActor après avoir été 'corrigées' par clBirdToCaptureMatrix de CSENSORS.
* stCentCalibTrans et stCentCalibRot (lignes 5 et 6) représentent, sous la forme de vecteurs, les matrices de calibration respectivement en rotation et en translation du capteur. clCentCalibRot et clCentCalibTrans (lignes 7 et 8) les représentent directement sous forme de matrice.
* pclAttachedModel (ligne 10) pointe vers le noeud de Model auquel le capteur est éventuellement attaché.
* dataFormat (ligne 11) détermine le format des données envoyées par le capteur: FLOCK_NOBIRDDATA, FLOCK_POSITION, FLOCK_ANGLES, FLOCK_MATRIX, FLOCK_POSITIONANGLES ou FLOCK_POSITIONMATRIX (voir le point A1.2.6.1).

msg □ msg est une variable ROOM de type ROOMMessage qui contient toujours le dernier message que l'acteur a reçu.

frameSAP □ frameSAP est un SAP qui permet à aMotionActor de communiquer avec le *frame service* ROOM. aMotionActor peur demander au *frame service* de créer un autre acteur en lui donnant un message d'initialisation.

A1.3.2.2. Etats

aMotionActor peut se trouver dans trois états différents :

- l'état Ready (l'état initial), dans lequel aMotionActor se trouve juste après s'être initialisé ;
- l'état Actor Loaded, dans lequel la structure Model contient un modèle 3D ;
- l'état Dead (l'état final), dans lequel aMotionActor termine son exécution.

A1.3.2.3. Transitions du ROOMChart et méthodes des classes CMODEL, COBJECT3D et CSENSORS

transition init :

```
1    action :
2    {      Sensors = new CSENSORS(msg->data);
3           Model = new CMODEL;
4           frameSAP.incarnate(anInterfaceActor, msg->data);
5    }
```

* aMotionActor est une instance optionnelle, incarnée par aDispatchActor. Lors de cette incarnation, aMotionActor reçoit un premier message, qui est copié dans msg et qui contient la configuration de aBirdsActor sous la forme d'une variable de type CONFIG (msg->data est un pointeur vers cette variable).
* msg->data permet d'initialiser Sensors (appel au constructeur de Sensors, voir ci-dessous), ligne 2. Model est également initialisé avec un modèle vide (appel au constructeur de Model, voir ci-dessous), ligne 3.
* aMotionActor crée ensuite anInterfaceActor (ligne 4), en lui donnant la configuration actuelle des capteurs. Pour faire cela, il fait appel à la méthode incarnate de son SAP frameSAP.

CSENSORS::CSENSORS(CONFIG currentConfig)

* L'interface de la classe CSENSORS est présentée au point A1.3.2.1; le type CONFIG est présenté au point A1.2.6.2.
* Le constructeur CSENSORS, sur base des champs sensorsNb et sampleRate de currentConfig, initialise iNbOfSensors et sampleRate. Il alloue ensuite de l'espace pour pstSensors selon le nombre de capteurs iNbOfSensors. Le champ dataFormat de chaque variable de type TSENSORDATA est initialisé en fonction du champ dataFormats du paramètre currentConfig. clBirdToCaptureMatrix est initialisée pour qu'elle représente la matrice de rotation entre le *frame* de aBirdsActor et le *frame* de openGLWin.

CMODEL::CMODEL()

* L'interface de la classe CMODEL est présentée au point A1.3.2.1.
* Le constructeur CMODEL crée un arbre contenant un seul noeud, dont le champ szNodeModelName est initialisé à root. CMODEL appelle aussi le constructeur de pclNodeObject.

transition t11 :

```
1   triggered by : {{load_model, pInterface}}
2   action :
3   {     Model->Load( (char*) msg->data);
4         Model->BoundingBox();
5         Model->Display();
6         Sensors->AttachSensor(1, "root", Model);
7   }
```

* t11 est franchie si anInterfaceActor envoie un message {load_model, char*}. aMotionActor, ligne 3, commence par initialiser Model avec les fichiers texte dont le nom du fichier principal est contenu dans le message.
* aMotionActor, ligne 4, calcule le volume englobant du modèle et l'envoie à anInterfaceActor pour que celui-ci puisse évaluer un espace virtuel assez important pour que le modèle 3D puisse s'y trouver quels que soient ses mouvements et ses déplacements. Sur base du volume englobant, anInterfaceActor détermine également la position de la caméra virtuelle fixant le modèle dans openGLWin.
* aMotionActor peut alors, ligne 5, envoyer les ordres nécessaires pour afficher le modèle dans openGLWin.
* aMotionActor attache ensuite le noeud 'root' du modèle avec le premier capteur.

void CMODEL::Load(char *szModel)

* Load, à partir d'un fichier texte principal d'extension '.def' et dont le nom est szModel, et à partir de fichiers secondaires d'extension '.obj', remplit Model.
  - Le fichier .def contient une liste de descriptions des noeuds du modèle, reprenant pour chaque noeud :
    - son nom,
    - le nom du fichier .obj dans lequel la description de l'objet 3D —main, tête, buste, etc— relatif au noeud est stockée, pour autant que le noeud ne soit pas un 'null';
    - la place du noeud dans l'arbre,
    - 3 valeurs représentant la matrice déterminant la position du *frame* local du noeud par rapport au *frame* du noeud père, et 3 valeurs représentant la matrice déterminant l'orientation de ce *frame* local par rapport au *frame* du noeud père.
  - Chaque fichier .obj contient les infos relatives à un objet 3D :
    - une liste des sommets définissant l'objet, un sommet étant défini par 3 valeurs réelles (coordonnées x, y et z) ;
    - une liste des vecteurs —un par sommet— nécessaires pour déterminer, selon les paramètres d'illumination de la scène, la couleur finale de la surface déterminée par les sommets, un vecteur étant défini par 3 valeurs réelles ;
    - une liste des faces (triangles ou quadrilatères) qui définissent la surface de l'objet. Cette liste détermine comment les sommets de la première liste participent à la définition des faces;
    - d'autres informations, comme la couleur ou la texture associée à la surface de l'objet, dont nous ne nous sommes pas servis. Nous l'aurions fait dans une étape ultérieure ; dans l'état dans lequel se trouve le système de capture de mouvements décrit dans ce travail, les modèles 3D sont affichés dans une couleur par défaut, unie.
* On trouvera, dans le code de GabbyCapture —dont une copie se trouve sur la disquette accompagnant le présent document—, des exemples de fichiers .def et .obj [1]. Load initialise tous les champs des noeuds de Model (matrices, vecteurs, booléens...)sur base des informations du fichier .def et envoie déjà une série d'informations à anInterfaceActor concernant le modèle :
  - chaque fois que Load insère un noeud dans Model, elle envoie son nom à anInterfaceActor,

[1] Le fichier Lisez-moi.txt qui se trouve dans le répertoire racine de la disquette détaille le contenu de celle-ci.

pour que celui-ci puisse l'ajouter dans ses listes qu'il montre à l'utilisateur (voir la classe d'acteur anInterfaceActor au point A1.4). Pour ce faire, Load exécute

pInterface.send(send_node_name, szNodeModelName);

qui lui permet d'envoyer le message {send_node_name, szNodeModelName} sur sa porte pInterface qui est reliée à anInterfaceActor.

- chaque fois que Load insère un noeud dans Model, elle appelle aussi la méthode Object3DLoad (ci-dessous) de l'objet 3D associé au noeud pour charger dans anInterfaceActor la représentation graphique de l'objet à partir du fichier .obj correspondant. En effet, comme nous le disions plus haut, les données graphiques associées au modèle 3D ne sont pas stockées dans Model, mais directement dans des *display lists* OpenGL.

void COBJECT3D::Object3DLoad(char *szFilename, int *iSensorDisplayListID, int *iDisplayListID)

* L'interface de la classe COBJECT3D est présentée au point A1.3.2.1.
* Object3DLoad, à partir des informations du fichier .obj de nom szFileName, construit deux *display lists* dans anInterfaceActor et renvoie à Model leurs identifiants (iSensorDisplayListID et iDisplayListID).

- Une première *display list* est créée pour stocker la représentation visuelle (un cube dont on ne voit que les arêtes) de l'éventuel capteur attaché à l'objet 3D. Le code qui suit implémente cette création.

```
1    {msg = pInterface.invoke(ask_list_id);
2     iSensorDisplayListID* = ((int*) msg->data)*;
3     pInterface.send(push_attribs);
4     pInterface.send(modif_drawcolor, drawColor);
5     pInterface.send(modif_drawline_width, 3.0);
6     pInterface.send(fill_list, iSensorDisplayListID*);
7          pInterface.send(begin_draw, GL_LINE_LOOP);
8               pInterface.send(send_vertex, vertex1);
9               pInterface.send(send_vertex, vertex2);
10              ...
11         pInterface.send(end_draw);
12         pInterface.send(begin_draw, GL_LINES);
13              pInterface.send(send_vertex, vertexa);
14              pInterface.send(send_vertex, vertexb);
15              ...
16         pInterface.send(end_draw);
17         ...
18    pInterface.send(stop_fill);
19    pInterface.send(pop_attribs);
20   }
```

• Après avoir demandé un identifiant de *display list* libre (ligne 1) et l'avoir stocké dans le paramètre résultat iSensorDisplayListID (ligne 2), Object3DLoad empile le jeu d'attributs OpenGL (ligne 3) car elle va modifier 2 attributs : la couleur de dessin et l'épaisseur du trait de dessin (lignes 4 et 5). drawColor est une variable de type COORD3 qui contient un code RGB.

• Object3DLoad remplit ensuite la *display list* (lignes 6 à 18) avec des formes géométriques de type GL_LINE_LOOP (lignes 7 à 11) et GL_LINES (lignes 12 à 16). Les variables vertex*x*, de type COORD3, désignent les sommets du cube.

• Object3DLoad dépile le jeu d'attributs (ligne 19).

- Une seconde *display list* est créée pour stocker l'objet 3D. Le code qui suit implémente cette création.

```
1    {msg = pInterface.invoke(ask_list_id);
2     iDisplayListID* = ((int*) msg->data)*;
3     pInterface.send(fill_list, iDisplayListID*);
4          pInterface.send(begin_draw, GL_TRIANGLES);
5               pInterface.send(send_normal,normal1); pInterface.send(send_vertex, vertex1);
```

```
6                   pInterface.send(send_normal,normal2); pInterface.send(send_vertex, vertex2);
7                   pInterface.send(send_normal,normal3); pInterface.send(send_vertex, vertex3);
8               pInterface.send(end_draw);
9               pInterface.send(begin_draw, GL_QUADS);
10                  pInterface.send(send_normal,normala); pInterface.send(send_vertex, vertexa);
11                  pInterface.send(send_normal,normalb); pInterface.send(send_vertex, vertexb);
12                  pInterface.send(send_normal,normalc); pInterface.send(send_vertex, vertexc);
13                  pInterface.send(send_normal,normald); pInterface.send(send_vertex, vertexd);
14              pInterface.send(end_draw);
15              ...
16          pInterface.send(stop_fill);
17  }
```

Après avoir demandé un identifiant de *display list* libre (ligne 1) et l'avoir stocké dans le paramètre résultat iDisplayListID (ligne 2), Object3DLoad remplit ensuite la *display list* (lignes 3 à 16) avec des formes géométriques de type GL_TRIANGLES (lignes 4 à 8) et GL_QUADS (lignes 9 à 14). Les variables vertex*x* *et* normal*x*, de type COORD3, désignent les sommets des surfaces des objets et leurs vecteurs.

* Object3DLoad calcule également le volume englobant de l'objet 3D, dont les sommets sont exprimés dans son *frame* local.

void CMODEL::BoundingBox()

* BoundingBox, sur base des volumes englobants de chaque objet 3D du modèle, calcule le volume dans lequel est contenu le modèle entier. Il stocke ce volume sous la forme de deux valeurs dans la variable stBoundingBox, de type COORD3[2], du champ pclNodeObject du noeud racine.
* BoundingBox envoie alors ces valeurs à anInterfaceActor en exécutant

pInterface.send(model_bbox, stBoundingBox);

void CMODEL::Display()

* Display envoie les ordres nécessaires à anInterfaceActor pour que celui-ci affiche le modèle dans sa fenêtre openGLWin. Puisque les données graphiques du modèle sont déjà stockées dans des *display lists* de openGLWin –selon la règle : une *display list* par objet du modèle–, aMotionActor doit juste envoyer à anInterfaceActor des messages de type {call_list, int}.

Cependant, les données graphiques relatives à un objet dans une *display list* sont exprimées dans le *frame* local de l'objet. Avant d'afficher celui-ci, il faut, pour qu'il apparaisse au bon endroit, exprimer ses sommets et ses vecteurs dans le *frame* global du modèle. Il faut pour cela :

- obtenir la matrice d'accumulation en rotation et en translation de l'objet, c'est-à-dire celle qui détermine la position et l'orientation du *frame* local de l'objet par rapport au *frame* global du modèle 3D ;
- multiplier les sommets et les vecteurs de l'objet par cette matrice avant de dessiner les formes géométriques qu'ils déterminent.

Sachant que anInterfaceActor, avant d'afficher une forme géométrique, multiplie ses sommets et ses vecteurs par sa matrice Modelview, il s'agit donc de remplir Modelview avec la matrice d'accumulation en rotation et en translation de l'objet, avant d'envoyer chaque message de type {call_list, int}. L'algorithme qui suit présente la manière dont Display envoie ses ordres à anInterfaceActor pour afficher le modèle.

```
1   { pInterface.send(clear);
2     pInterface.send(select_matrix, GL_MODELVIEW);
3     _Display();
4     pInterface.send(swap_buffers);
5   }
```

- Display nettoie d'abord le *back buffer* de openGLWin dans lequel elle s'apprête a dessiner (ligne 1). Elle rend ensuite courante la matrice Modelview (ligne 2) avant d'exécuter la méthode récursive _Display (ci-dessous), qui réalise une recherche en profondeur d'abord de l'arbre.
- Display termine en intervertissant les *buffers* de openGLWin pour rendre visible le modèle qu'elle vient de dessiner dans le *back buffer*.

```
void CMODEL::_Display()
1    { CMATRIX tr;
2      pInterface.send(push_matrix);
3          tr.Load(pclNodeObject->clTrans);
4          tr.MultRightMatrix(pclNodeObject->clRot);
5          pInterface.send(mult_right_matrix, tr);
6          if (iDisplayListID != -1) pInterface.send(call_list, iDisplayListID);
7          if (pstSensordata != NULL) pInterface.send(call_list, iSensorDisplayListID);
8          if (pclSons != NULL)
9                pclSons->Display();
10         pInterface.send(pop_matrix);
11         if (pclNext != NULL)
12               pclNext->Display();
13   }
```

- _Display crée 1 nouvelle copie de Modelview sur la pile des matrices Modelview (ligne 2). A ce moment, Modelview contient toujours la matrice d'accumulation (rotation et translation) du noeud père. La première fois que _Display est exécutée, Modelview est égale à la matrice identité.
- Lignes 3 et 4 : _Display calcule, sur base des 2 matrices déterminant la position et l'orientation du frame local du noeud par rapport au frame du noeud père, une matrice unique (tr) qui détermine à la fois la position et l'orientation du *frame* local du noeud par rapport à son père.
- En multipliant Modelview par la matrice qu'on a calculée ligne 3, on obtient la matrice d'accumulation en rotation et translation pour le noeud courant (ligne 4).
- _Display peut alors dessiner l'objet, ligne 5 (pour autant que le noeud courant ne soit pas un 'null'), et l'éventuel capteur attaché à cet objet (ligne 6).
- _Display s'exécute ensuite récursivement sur les fils du noeud courant, puis sur ses frères. Pour que la matrice d'accumulation soit toujours, par rapport au noeud traité, celle du noeud père, avant de traiter un frère, il faut dépiler une matrice sur la pile des matrices Modelview.

* La sémantique de CMATRIX::Load est : A.Load(B) ⇔ A ← B ;
La sémantique de CMATRIX::MultRightMatrix est : A.MultRightMatrix(B) ⇔ A ← A * B .
← symbolise l'assignation.
pInterface.send(mult_right_matrix, tr) a l'effet suivant : Modelview ← Modelview * tr.

void CSENSORS::AttachSensor(int iSensorId, char *szNodeModelName, CMODEL *Model)

* AttachSensor a pour but d'attacher le capteur de n° iSensorId au noeud de Model de nom szNodeModelName.
* Pour ce faire, AttachSensor détermine, au sein de l'objet Sensors, l'adresse de la structure TSENSORDATA relative au capteur de n° iSensorId. Ensuite, elle appelle la méthode AttachThisSensor de l'objet Model en lui donnant comme arguments l'adresse de la structure TSENSORDATA relative au capteur à attacher et le nom du noeud à attacher. AttachThisSensor renvoie l'adresse du noeud sur lequel le capteur a été attaché, ce qui permet à AttachSensor de mettre à jour l'objet Sensors, ou, pour être précis, de mettre à jour le pointeur pclAttachedModel de la structure TSENSORDATA relative au capteur de n° iSensorID.
* AttachSensor appelle ensuite la méthode UpdateSensorsFlags de Model.

CMODEL *CMODEL::AttachThisSensor(TSENSORDATA *sensorData, char *szNodeModelName)

* AttachThisSensor parcourt l'arbre Model jusqu'à ce qu'elle trouve un noeud dont le nom est szNodeModelName, met à jour la structure pstSensorData relative à ce noeud –elle "attache" le capteur dont l'adresse est sensorData– et renvoie l'adresse du noeud sur lequel elle a effectué cette opération.

void CMODEL::UpdateSensorsFlags()

* UpdateSensorsFlags a pour objectif de mettre à jour les booleens bSonsHaveSensors et bNextHaveSensors. Elle est appelée à chaque fois que l'on attache ou que l'on détache un capteur de Model.

transition t12 :

```
1   triggered by : { {modif_config, pInterface} or {calib_trans, pInterface} or {calib_rot, pInterface} }
2   action :
3   {     switch (msg->signal)
4         {     case modif_config :
5                     Sensors->ModifConfig(msg->data);
6                     pDispatch.send(modif_config, msg->data);
7                     break;
8               case calib_trans :
9                     Sensors->UpdateCalibrationTrans(msg->data);
10                    break;
11              case calib_rot :
12                    Sensors->UpdateCalibrationRot(msg->data);
13                    break;
14        }
15  }
```

* Quand il reçoit de anInterfaceActor une modification de configuration, aMotionActor la répercute sur son objet Sensors (ligne 5) et l'envoie aussi à aDispatchActor (ligne 6).
* Quand aMotionActor reçoit une nouvelle matrice de calibration en translation (rotation) relative à un capteur donné, il exécute la méthode UpdateCalibrationTrans (UpdateCalibrationRot), ligne 9 (ligne 12) pour mettre à jour Sensors.

void CSENSORS::ModifConfig(CONFIG currentConfig)

* ModifConfig, sur base des champs sampleRate et dataFormats du paramètre currentConfig, met à jour sampleRate et la variable dataFormat pour chaque capteur de Sensors. Le nombre de capteurs ne peut pas être modifié.

void CSENSORS::UpdateCalibrationTrans(CALIB calib)

* UpdateCalibrationTrans a pour but de mettre à jour, sur base de calib.calibdata (de type COORD3) les variables stCentCalibTrans et clCentCalibTrans relatives au capteur dont le numéro est donné par calib.sensorId.

void CSENSORS::UpdateCalibrationRot(CALIB calib)

* UpdateCalibrationRot a pour but de mettre à jour, sur base de calib.calibdata (de type COORD3) les variables stCentCalibRot et clCentCalibRot relatives au capteur dont le numéro est donné par calib.sensorId.

transition t13 :

```
1   triggered by : { {kill, pInterface} }
2   action : -
```

* Quand anInterfaceActor signifie à aMotionActor que le système doit être clôturé, aMotionActor passe dans l'état Dead. Aucune action n'est associée à t13.

transition t21 :

```
1   triggered by : { {delete_model, pInterface} }
2   action :
    {     deleteModel;
          newModel;
    }
```

* Quand il reçoit un message lui enjoignant d'effacer Model, aMotionActor appelle le destructeur de Model, puis crée un nouveau modèle vide en appelant le constructeur de Model.

CMODEL::~CMODEL()

* ~CMODEL appelle le destructeur de pclNodeObject, pclNext et pclSons. Pour chaque capteur attaché au modèle, il appelle la méthode DetachSensor de l'objet Sensors.

void CSENSORS::DetachSensor(int iSensorId, CMODEL *Model)

* DetachSensor a pour but de détacher le capteur de n° iSensorId du noeud de Model auquel il est attaché. Puisque la variable pclAttachedModel du capteur iSensorID de l'objet Sensors pointe sur le noeud auquel est attaché ce capteur, DetachSensor met directement à jour ce noeud en mettant à NULL sa variable pstSensorData (il 'détache' le noeud). DetachSensor détache ensuite le capteur en mettant son champ pclAttachedModel à NULL.
* DetachSensor appelle ensuite la méthode UpdateSensorsFlags de l'objet Model.

transition t22 :

```
1   triggered by : { {redisplay_model, pInterface} or {attach, pInterface} or {detach, pInterface} or
2                    {modif_config, pInterface} or {calib_trans, pInterface} or {calib_rot, pInterface}
3                    or {start, pInterface} or {stop, pInterface} or {send_data, pStream} }
4   action :
5   {     switch (msg->signal)
6         {     case redisplay_model :
7                     Model->Display();
8                     break;
9               case attach :
10                    Sensors->AttachSensor( (int) msg->data[0], &((char) msg->data[1]), Model);
11                    Model->Display();
12                    break;
13              case detach :
14                    Sensors->DetachSensor( ((int*) msg->data)*, Model);
15                    Model->Display();
16                    break;
17              case modif_config :
18                    Sensors->ModifConfig(msg->data);
19                    pDispatch.send(modif_config, msg->data);
20                    break;
21              case calib_trans :
22                    Sensors->UpdateCalibrationTrans(msg->data);
23                    break;
24              case calib_rot :
25                    Sensors->UpdateCalibrationRot(msg->data);
26                    break;
27              case start :
28                    pDispatch.send(start);
29                    break;
30              case stop:
31                    pDispatch.send(stop);
32                    break;
33              case send_data:
34                    Sensors->UpdateDataMatrices(msg->data);
35                    Model->UpdateLocalTransfos();
36                    Model->Display();
37                    break;
38        }
39  }
```

* Quand il reçoit un message lui enjoignant de réafficher le modèle, aMotionActor appelle la méthode Display de Model (lignes 6 à 8).
* Quand il reçoit un message lui enjoignant d'attacher ou de détacher un noeud du modèle et un capteur, il appelle respectivement les méthodes AttachSensor et DetachSensor de l'objet

Sensors (lignes 9 à 16).
* Les réactions de aMotionActor par rapport à la réception de messages de type {modif_config, CONFIG}, {calib_trans, CALIB} et {calib_rot, CALIB} (lignes 17 à 26), alors qu'il est dans l'état Actor Loaded (le cas qui nous occupe ici) sont les mêmes que lorsqu'il se trouve dans l'état Ready (voir la transition t12).
* Quand il reçoit un message lui enjoignant de lancer ou de stopper l'animation, aMotionActor transmet le message à aDispatchActor (lignes 27 à 32).
* Quand aMotionActor reçoit de aStreamActor un message de données (ligne 33), il met d'abord à jour l'objet Sensors (appel à la méthode UpdateDataMatrices, ligne 34). Sur base de ce dernier et des liens qu'il existe entre les capteurs et les objets du modèle, UpdateLocalTransfos (ligne 35) met à jour Model. Celui-ci doit alors être réaffiché (appel à la fonction Display, ligne 36).

void CSENSORS::UpdateDataMatrices(char* data)

* UpdateDataMatrices met à jour les matrices clDataTrans et clDataRot de chaque capteur de Sensors avec les données relatives aux capteurs qui lui sont fournies dans le paramètre data.
* Chaque groupe de 8 caractères contigüs, dans data, représente une valeur réelle faisant partie soit d'un groupe de 3 valeurs mesurées en pouces, soit d'un groupe de 3 valeurs mesurées en degrés, soit encore d'une matrice de rotation. UpdateDataMatrices connaît le type de chaque valeur, sait de quel vecteur ou matrice elle fait partie, et quel capteur l'a générée, grâce au champ dataFormat de chaque capteur dans Sensors. La structure précise de data sera abordée sur la figure A1.21 (à droite sur la figure).

void CMODEL::UpdateLocalTransfos()

* UpdateLocalTransfos met à jour les matrices des noeuds de Model auxquels sont attachés des capteurs pour faire adopter au modèle la même posture que l'être humain qui a les capteurs sur lui. UpdateLocalTransfos est implémentée comme suit.

```
1    { pclNodeObject->clTrans.Load(pstSensorData->clDataTrans);
2      pclNodeObject->clTrans.MultRightMatrix(pstSensorData->clCentCalibTrans);
3      pclNodeObject->clRot.Load(pstSensorData->clDataRot);
4      pclNodeObject->clRot.MultRightMatrix(pstSensorData->clCentCalibRot);
5      clAccumRot.Load(pclNodeObject->clRot);
6      pclSons->_UpdateLocalTransfos();
7    }
```

- UpdateLocalTransfos est toujours appelée sur le noeud 'root' de Model. Elle met donc à jour la matrice pclNodeObject->clTrans déterminant la position du noeud 'root' sur base de la matrice déterminant la position du capteur qui lui est associé et de la matrice de calibration en translation de ce capteur (lignes 1 et 2).
- Elle met ensuite à jour la matrice pclNodeObject->clRot déterminant l' orientation du noeud 'root' sur base de la matrice déterminant l'orientation du capteur qui lui est associé et de la matrice de calibration en rotation de ce capteur (lignes 3 et 4).
- Elle calcule, ligne 5, la matrice d'accumulation en rotation du noeud 'root', qui n'est autre que la matrice déterminant l'orientation du noeud 'root', après avoir été mise à jour par les lignes 3 et 4.
- Elle appelle ensuite la méthode récursive _UpdateLocalTransfos qui met à jour les autres noeuds de Model.

* La sémantique de CMATRIX::Load est : A.Load(B) ⇔ A ← B ;
La sémantique de CMATRIX::MultRightMatrix est : A.MultRightMatrix(B) ⇔ A ← A * B .
← symbolise l'assignation.

void CMODEL::_UpdateLocalTransfos()

```
1    {
2      CMATRIX clInverseFatherAccumRot;
3
4      if (pstSensorData != NULL)
5      {   clInverseFatherAccumRot.Load(pclFather->clAccumRot);
6          clInverseFatherAccumRot.Invert();
```

```
7           pclNodeObject->clRot.Load(clInverseFatherAccumRot);
8           pclNodeObject->MultRightMatrix(pstSensorData->clDataRot);
9           pclNodeObject->MultRightMatrix(pstSensorData->clCentCalibRot);
10      }
11      clAccumRot->Load(pclFather->clAccumRot);
12      clAccumRot->MultRightMatrix(pclNodeObject->clRot);
13      if (pclSons != NULL) pclSons->_UpdateLocalTransfos();
14      if (pclNext != NULL)  pclNext->_UpdateLocalTransfos();
15  }
```

* Si un capteur est attaché au noeud courant :
1• _UpDateLocalTransfos calcule la matrice d'accumulation en rotation, inversée, de son noeud père (clInverseFatherAccumRot), lignes 5 et 6;
2• elle met à jour la matrice pclNodeObject->clRot déterminant l'orientation du noeud courant sur base de la matrice de clInverseFatherAccumRot, de la matrice déterminant l'orientation du capteur qui lui est associé et de la matrice de calibration en rotation de ce capteur (lignes 7 à 9) ;
3• elle calcule la matrice d'accumulation en rotation du noeud courant (lignes 11 et 12);
4• elle s'exécute ensuite sur les fils (ligne 13) et les frères (ligne 14) du noeud courant.
* Si aucun capteur n'est attaché au noeud courant, elle n'effectue que les étapes 3 et 4 ci-dessus.

transition t23 :

```
1   triggered by : {{kill, pInterface}}
2   action : -
```

* Quand anInterfaceActor signifie à aMotionActor que le système doit être clôturé, aMotionActor passe dans l'état Dead. Aucune action n'est associée à t23.

entry action kill :

```
1   {       delete Model;
2           delete Sensors;
3           pDispatch.send(kill);
4   }
```

* aMotionActor, avant de terminer son exécution, appelle les destructeurs de Model et de Sensors (lignes 1 et 2) et transmet le message de clôture à aDispatchActor (ligne 3).

A1.4. La classe d'acteur 'InterfaceActor'

La section A1.4 présente, de manière informelle, la structure et le comportement de la classe InterfaceActor, dont l'unique instance anInterfaceActor constitue l'interface utilisateur de notre système de capture de mouvements.

A1.4.1. Structure

La structure de la classe InterfaceActor est présentée à la figure A1.7.

Toute instance de InterfaceActor dispose d'une porte lui permettant de communiquer selon le protocole pMotionInterface et d'une porte lui permettant de communiquer avec l'utilisateur du système. Dans notre modèle, la porte pMotion de la seule instance anInterfaceActor est reliée à aMotionActor, tandis que la porte pUser est «reliée» à l'utilisateur.

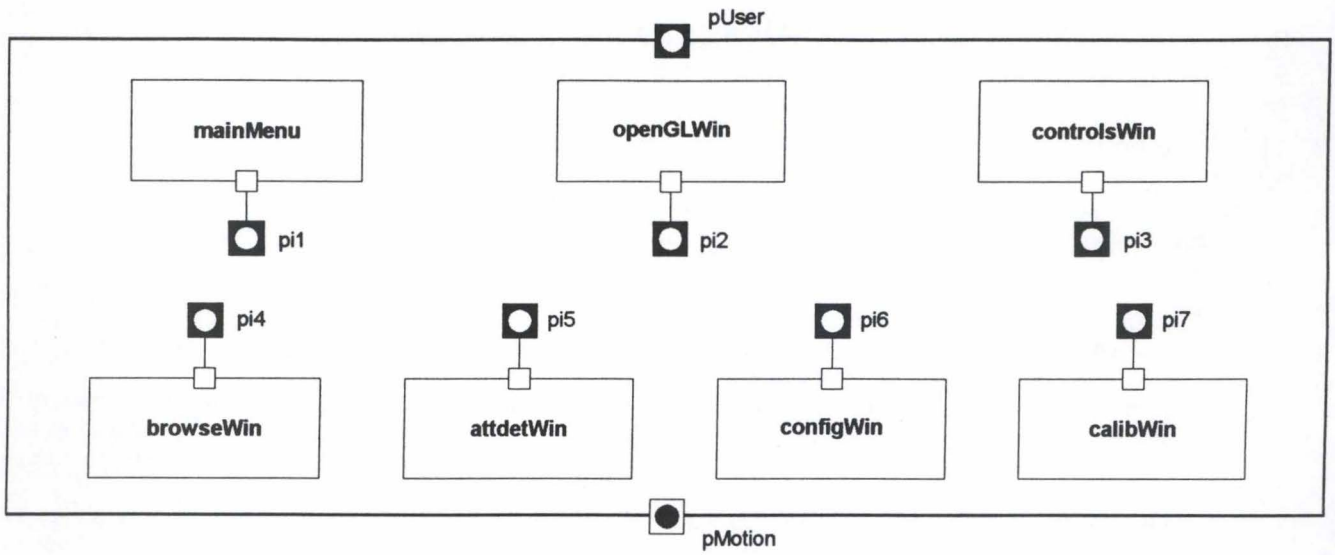


Figure A1.7 Structure de InterfaceActor

Les portes intérieures pi1 à pi7 sont utilisées par anInterfaceActor pour communiquer avec les acteurs qui sont contenus à l'intérieur de lui-même. Ceux-ci sont au nombre de 7. Leur structure est examinée ci-dessous (points A1.4.1.1 à A1.4.1.7). A l'instar de anInterfaceActor, ils disposent tous d'une représentation graphique.

A1.4.1.1. mainMenu

mainMenu, de la classe MENU, est une barre de menus déroulants contenant deux menus :
- le menu ' *File* ' contient les items ' *Open* ', ' *Close* ' et ' *Quit* ' ;
- le menu ' *Sensor* ' contient les items ' *Attach/Detach* ', ' *Configure* ' et ' *Calibrate* '.

A1.4.1.2. openGLWin

openGLWin, de la classe OPENGLWINDOW, est la fenêtre dans laquelle le modèle 3D est affiché. Elle contient notamment les variables suivantes, dont les 6 premières correspondent à celles d'OpenGL:

1• une pile de matrices Modelview ;
2• une pile de matrices Projection ;
3• un booléen selectedMatrix indiquant quelle est la pile courante ;
4• une pile d'attributs attPile ;
5• une structure displaylists contenant un ensemble de *display lists* ;
6• une variable buffers contenant les *back* et *front buffers*.
7• un booléen modelLoaded qui est TRUE si un modèle est chargé dans l'objet Model de aMotionActor;
8• deux entiers width et length contenant la largeur et la longueur (en pixels) d'openGLWin.

A1.4.1.3. controlsWin

controlsWin, de la classe CONTROLDLG, permet à l'utilisateur de lancer ou de stopper l'animation et de de manipuler la caméra virtuelle qui «filme» le modèle dans openGLWin. controlsWin

contient notamment :

- un booléen transfType déterminant le type de transformation (translation ou rotation) qui est appliqué à caméra «filmant» le modèle dans openGLWin quand l'utilisateur décide, en faisant glisser le curseur de la souris sur openGLWin avec un bouton enfoncé, de déplacer la caméra ;
- un entier axis déterminant l'axe de rotation (ou de translation) autour duquel tourne cette caméra quand l'utilisateur la déplace ;
- des boutons ' *start* ' et ' *stop* '.

Les représentations graphiques de anInterfaceActor, de mainMenu, d'openGLWin et de controlsWin font l'objet de la figure A1.8. Un cyberacteur-test répondant au nom d'Oscar est affiché dans openGLWin. Les cubes verts dessinés sur certains de ses membres signalent que des capteurs y sont associés.

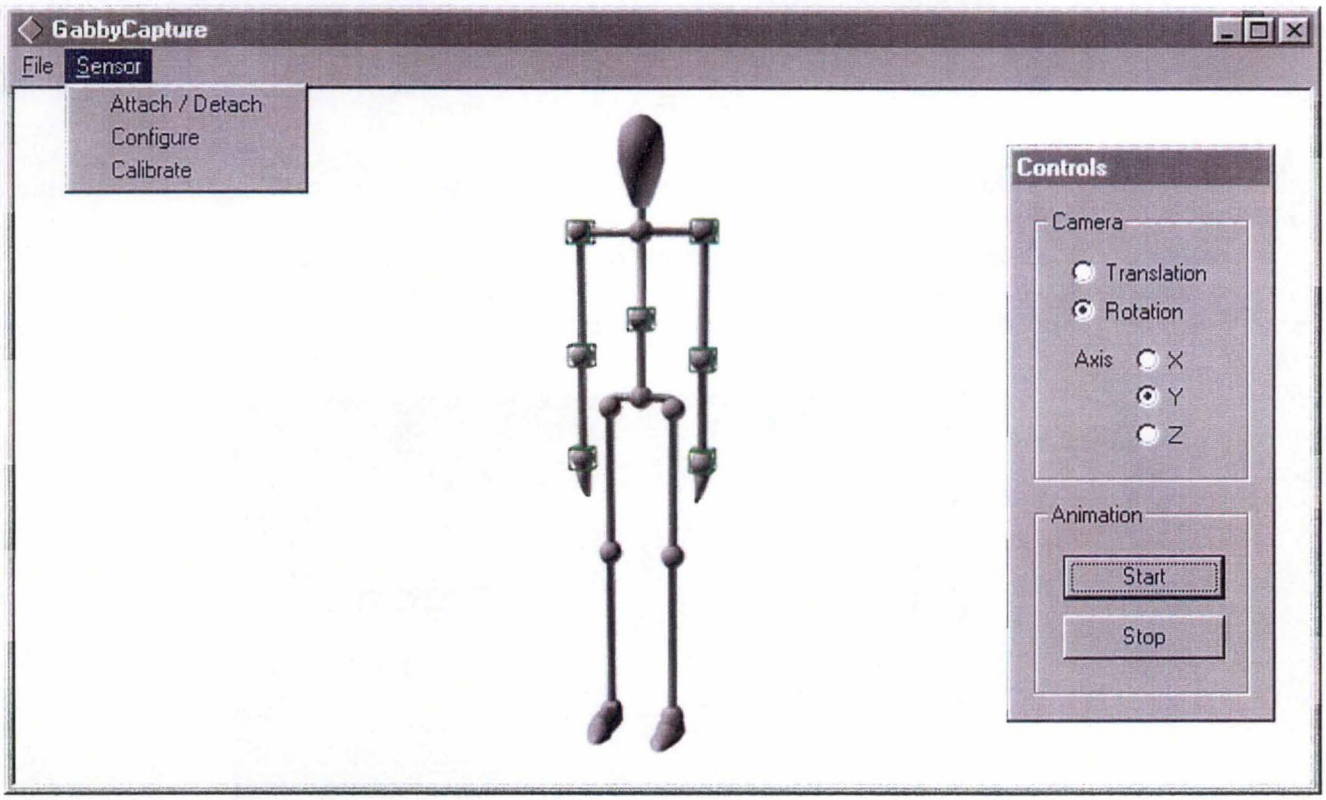


Figure A1.8
Représentation graphique de anInterfaceActor, mainMenu, openGLWin et controlsWin

A1.4.1.4. browseWin

browseWin, de la classe BROWSEDLG, permet de parcourir un catalogue de fichiers et de répertoires et contient entre autres une variable pouvant contenir un tel catalogue. La figure A1.9 présente sa représentation graphique.

A1.4.1.5. attdetWin

attdetWin, de la classe ATTDETDLG, est représentée sur la figure A1.10. Elle permet à l'utilisateur

Figure A1.9 browseWin

d'associer un à un (ou de dissocier) les capteurs et les membres du modèle 3D. Elle contient notamment:

- une liste actorHierarchy reprenant les noms de tous les noeuds du modèle,
- une liste sensors reprenant les numéros de tous les capteurs de aBirdsActor,
- des boutons ' << *Attach* ', ' << *Detach* ' et ' *Close* '.

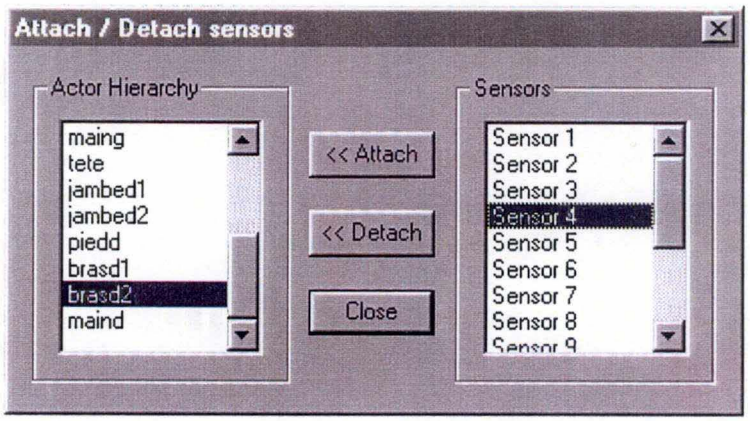


Figure A1.10 attdetWin

A1.4.1.6. configWin

configWin permet à l'utilisateur de modifier la configuration de aBirdsActor : nombre de messages envoyés par seconde par les capteurs et format des données envoyées par chaque capteur. configWin est de la classe CONFIGDLG et est représentée sur la figure A1.11. Elle contient notamment:

- une liste sensors reprenant les numéros de tous les capteurs de aBirdsActor,
- une liste dataTypes reprenant les formats de données possibles,
- deux booléens, modifyAll et modifiedConfig, dont la signification sera donnée au point A1.4.2.8,
- un entier sampleRate contenant le nombre de messages par seconde envoyés par les capteurs,
- une variable currentConfig de type CONFIG donnant, pour chaque capteur, le format des données qu'il génère,
- un bouton ' *Close* '.

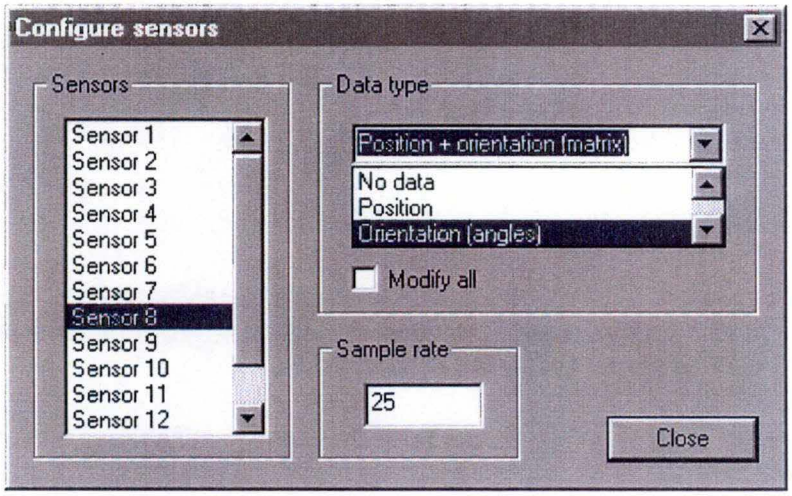


Figure A1.11 configWin

A1.4.1.7. calibWin

calibWin, de la classe CALIBDLG (figure A1.12), fournit un moyen à l'utilisateur de modifier les matrices de calibration en translation et en rotation de chaque capteur. Elle contient notamment :

- une liste sensors reprenant les numéros de tous les capteurs de aBirdsActor,
- un tableau transCalib qui contient autant d'éléments de type COORD3 qu'il n'y a de capteurs,
- des boutons ' + ' et ' - ' qui permettent de modifier les éléments de transCalib,
- un tableau rotCalib qui contient autant d'éléments de type COORD3 qu'il n'y a de capteurs,
- des réglettes qui permettent de modifier les éléments de transCalib,
- un tableau sensorsNodes qui, pour chaque capteur, contient le nom du noeud de Model auquel il est attaché,
- une variable attachedNode qui contient le nom du noeud de Model auquel est attaché le capteur qui est sélectionné dans la liste sensors,
- un bouton ' *Close* '.

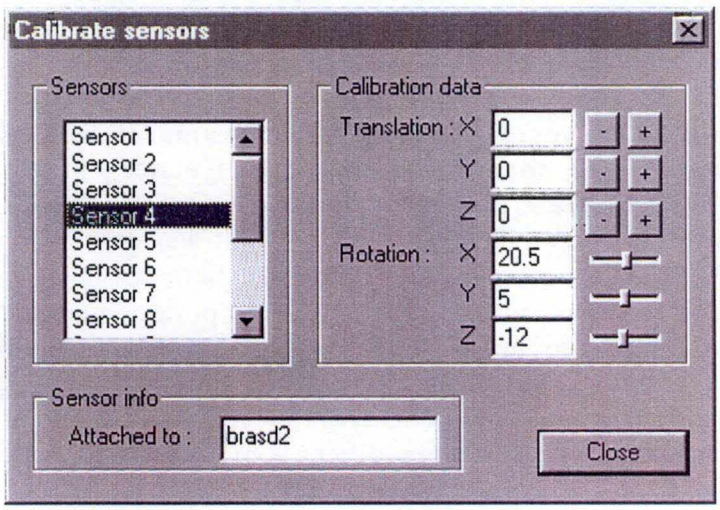


Figure A1.12 calibWin

A1.4.2. Comportement

Les points qui suivent déterminent le comportement de anInterfaceActor lors de la survenance de certains événements.

A1.4.2.1. 'aMotionActor' crée 'anInterfaceActor'

anInterfaceActor est une instance optionnelle, incarnée par aMotionActor. Lors de cette incarnation, anInterfaceActor reçoit un premier message qui contient la configuration de aBirdsActor sous la forme d'une variable de type CONFIG.

- anInterfaceActor, mainMenu, openGLWin et controlsWin apparaissent à l'écran. Les *frame buffers* d'openGLWin sont tous les deux «peints» en blanc. transfType de controlsWin est mise à 1 (rotation) et axis est mis à 1 (axe *y*).
- Les listes sensors de attdetWin, configWin et calibWin sont initialisées sur base du nombre de capteurs, information qui provient de aMotionActor.
- la variable de type CONFIG qui provient de aMotionActor est copiée dans currentConfig de configWin, sampleRate est initialisée sur base de currentConfig.sampleRate et dataTypes est initialisée avec les formats de données possibles pour les capteurs : aucune donnée, position uniquement, orientation uniquement, position et orientation, etc.
- sensorNode[0] de calibWin, qui contient le nom du noeud attaché au capteur 1, est mis à 'root'.

A1.4.2.2. L'utilisateur demande d'afficher un nouveau modèle

Quand l'utilisateur appuie sur l'item ' *Open* ' du menu ' *File* ' de mainMenu, qu'il fait ainsi apparaître browseWin, qu'il sélectionne dans le catalogue un fichier '.def' et qu'il appuie sur ' *Ok* ' :

- browseWin disparaît,
- anInterfaceActor envoie à aMotionActor un message de type {load_model, char*}, la variable de type char* contenant le nom du fichier '.def'. Si la variable modelLoaded de openGLWin est TRUE, anInterfaceActor envoie au préalable un message de type {delete_model}.
- Quand anInterfaceActor reçoit un message {send_node_name, char*}, il met à jour actorHierarchy de attdetWin.
- Quand elle reçoit une demande d'identifiant de *display list* {ask_list_id}, elle répond par un {list_id, int}, la variable de type int contenant un identifiant de *display list* libre. Les autres messages qui sont envoyés par aMotionActor chaque fois que celui-ci exécute la méthode Object3DLoad sur un noeud de Model —{push_attribs}, {modif_draw_color, COORD3}, {fill_list, int}, {begin_draw, int}, {send_vertex, COORD3}...— entraînent des modifications dans les variables de openGLWin (displaylists, attPile, etc).
- Quand elle reçoit un message de type {model_bbox, BBOX} :
  - elle évalue un espace virtuel assez important pour que le modèle 3D puisse s'y trouver quels que soient ses mouvements et ses déplacements. En d'autres termes, elle évalue le *clipping volume* ou la portion du monde virtuel qui pourra faire partie des images filmées par la «caméra virtuelle» d'openGLWin. Elle le fait en modifiant la matrice Projection ;
  - elle détermine également, en modifiant la matrice Modelview, la position de la caméra virtuelle fixant le modèle dans openGLWin ;
- Quand elle reçoit les messages de aMotionActor alors que celui-ci exécute la méthode Display de l'objet Model —{clear}, {push_matrix}, {call_list, int}, {swap_buffers}...—,

anInterfaceActor modifie en conséquence les variables de openGLWin : le modèle s'affiche à l'écran.

L'utilisateur ne peut pas appuyer sur l'item ' *Open* ' du menu ' *File* ' de mainMenu quand le modèle est en train d'être animé.

A1.4.2.3. L'utilisateur demande d'effacer le modèle affiché

Quand l'utilisateur appuie sur l'item ' *Close* ' du menu ' *File* ' de mainMenu, pour autant que la variable modelLoaded de openGLWin soit à TRUE :

- anInterfaceActor envoie à aMotionActor un message de type {delete_model},
- anInterfaceActor réinitialise toutes les variables de openGLWin : elle vide ses piles de matrices et d'attributs, elle remplit en blanc ses *buffers*, etc.

L'utilisateur ne peut pas appuyer sur l'item ' *Close* ' du menu ' *File* ' de mainMenu quand le modèle est en train d'être animé.

A1.4.2.4. L'utilisateur demande de terminer l'exécution du système de capture de mouvements

Quand l'utilisateur appuie sur l'item ' *Quit* ' du menu ' *File* ' de mainMenu, ou qu'il demande de terminer l'exécution du système par un autre moyen, anInterfaceActor envoie un message {kill} à aMotionActor avant de terminer sa propre exécution.

L'utilisateur ne peut pas demander de terminer l'exécution du système de capture de mouvements quand le modèle est en train d'être animé.

A1.4.2.5. 'openGLWin' est masquée par une autre fenêtre puis est de nouveau visible

Quand une partie (ou la totalité) d'openGLWin est masquée puis est exposée à nouveau, pour autant que la variable modelLoaded de openGLWin soit à TRUE :

- anInterfaceActor envoie à aMotionActor un message de type {redisplay_model},
- quand elle reçoit les messages de aMotionActor alors que celui-ci exécute la méthode Display de l'objet Model elle modifie en conséquence les variables de openGLWin : le modèle s'affiche à l'écran.

A1.4.2.6. L'utilisateur fait glisser la souris sur openGLWin avec un bouton de la souris enfoncé

Quand l'utilisateur fait glisser la souris sur openGLWin avec un bouton de la souris enfoncé, pour autant que la variable modelLoaded de openGLWin soit à TRUE :

- anInterfaceActor, sur base des variables transType et axis de controlsWin, modifie Modelview de openGLWin : la caméra virtuelle de openGLWin effectue le mouvement voulu par l'utilisateur ;
- anInterfaceActor envoie à aMotionActor un message de type {redisplay_model}.

A1.4.2.7. Actions de l'utilisateur concernant 'attdetWin'

Quand l'utilisateur appuie sur l'item ' *Attach/Detach* ' du menu ' *Sensor* ' de mainMenu, attdetWin apparaît ; quand il appuie sur le bouton ' *Close* ', elle disparaît. L'utilisateur peut toujours manipuler les autres fenêtres durant le temps où attdetWin est affichée à l'écran.

Quand l'utilisateur appuie sur le bouton ' << *Attach* ', pour autant
- qu'il ait sélectionné un item dans chaque liste (actorHierarchy et sensors),
- que ces items ne sont pas 'root' dans actorHierarchy et 'Sensor 1' dans sensors,
- que ces items ne participent pas déjà à une association,

anInterfaceActor « attache » le capteur sélectionné dans sensors avec le noeud (du modèle) sélectionné dans actorHierarchy :
- elle envoie à aMotionActor un message de type {attach, char*}, le premier caractère de la variable de type char* contenant le numéro du capteur et le reste des caractères contenant le nom du noeud de Model ;
- elle met à jour sensorsNodes de calibWin.

Quand l'utilisateur appuie sur le bouton ' << *Detach* ', pour autant
- qu'il ait sélectionné un item dans chaque liste,
- que ces items ne sont pas 'root' et 'Sensor 1',
- que ces items participent déjà à une association,

anInterfaceActor « détache » le capteur sélectionné dans sensors avec le noeud (du modèle) sélectionné dans actorHierarchy :
- elle envoie à aMotionActor un message de type {detach, int}, la variable de type int contenant le numéro du capteur à dissocier ;
- elle met à jour sensorsNodes de calibWin.

Quand l'utilisateur sélectionne un noeud dans actorHierarchy ou un capteur dans sensors, pour autant qu'il fasse partie d'une association, anInterfaceActor sélectionne dans l'autre liste l'élément auquel il est associé.

A1.4.2.8. Actions de l'utilisateur concernant 'configWin'

Quand l'utilisateur appuie sur l'item ' *Configure* ' du menu ' *Sensor* ' de mainMenu, configWin apparaît. L'utilisateur peut toujours manipuler les autres fenêtres durant le temps où configWin est affichée à l'écran.

Quand l'utilisateur sélectionne un capteur dans sensors, son format de données apparaît dans dataTypes. Quand il coche la case ' *Modify all* ', modifyAll est mis à TRUE.

Quand l'utilisateur modifie le format de données dans dataTypes,

- si modifyAll est à FALSE (sa représentation graphique n'est pas cochée), anInterfaceActor modifie la configuration du capteur sélectionné avec le nouveau format de données :
  - elle met à jour currentConfig de configWin,
  - elle n'envoie pas encore de message à aMotionActor pour lui communiquer la modification,
  - elle met à TRUE modifiedConfig s'il n'était pas déjà dans cet état : celui-ci détermine donc si des modifications de configuration sont intervenues depuis la dernière ouverture de

configWin.

- Si modifyAll est à TRUE (sa représentation graphique est cochée), anInterfaceActor applique le nouveau format de données sur *tous* les capteurs. L'action de anInterfaceActor est similaire à celle spécifiée pour un seul capteur (ci-dessus).

Quand l'utilisateur modifie sampleRate (dont la représentation visuelle est un champ d'édition), anInterfaceActor met à jour currentConfig, puis modifiedConfig si celui-ci n'était pas déjà à TRUE.

Quand l'utilisateur appuie sur le bouton ' *Close* ', configWin disparaît, et si modifiedConfig est à TRUE, anInterfaceActor envoie un message de type {modif_config, CONFIG} à aMotionActor, la variable de type CONFIG contenant currentConfig de configWin. modifiedConfig est remis à FALSE.

A1.4.2.9. Actions de l'utilisateur concernant 'calibWin'

Quand l'utilisateur appuie sur l'item ' *Calibrate* ' du menu ' *Sensor* ' de mainMenu, calibWin apparaît. Quand l'utilisateur appuie sur le bouton ' *Close* ', elle disparaît. L'utilisateur peut toujours manipuler les autres fenêtres durant le temps où calibWin est affichée à l'écran.

Quand l'utilisateur sélectionne un capteur dans sensors, ses matrices de translation et de rotation sont affichées dans les champs d'édition constituant la représentation visuelle de transCalib et rotCalib, sous la forme de 3 valeurs (une par axe). anInterfaceActor, sur base de sensorsNodes, détermine la valeur de attachedNode.

Quand l'utilisateur appuie sur un des boutons ' + ' ou ' - ', ou quand il va directement modifier les valeurs des champs d'édition de transCalib,

- transCalib est mis à jour avec la nouvelle valeur (le bouton ' + ' (' - ') incrémente (décrémente) une valeur de 0,1 pouce),
- anInterfaceActor envoie à aMotionActor un message de type {calib_trans, CALIB}. La variable de type CALIB contient le numéro du capteur ainsi que ses 3 valeurs de calibration en translation.

Quand l'utilisateur manipule une des réglettes, ou quand il va directement modifier les valeurs des champs d'édition de rotCalib,

- rotCalib est mis à jour avec la nouvelle valeur,
- anInterfaceActor envoie à aMotionActor un message de type {calib_rot, CALIB}. La variable de type CALIB contient le numéro du capteur ainsi que ses 3 valeurs de calibration en rotation.

A1.4.2.10. Actions de l'utilisateur concernant 'controlsWin'

controlsWin est toujours visible. En manipulant les boutons radio qui constituent les représentations visuelles de transfType et axis, l'utilisateur peut modifier ces dernières. Quand l'utilisateur appuie sur le bouton ' *start* ' (' *stop* '), pour autant que modelLoaded de openGLWin soit à TRUE, anInterfaceActor envoie à aMotionActor un message de type {start} ({stop}).

Quand anInterfaceActor reçoit les messages de aMotionActor alors que celui-ci exécute la méthode Display de l'objet Model, anInterfaceActor modifie en conséquence les variables de openGLWin : le modèle s'affiche à l'écran.

A1.5. La classe d'acteur 'BirdsActor'

La section A1.5 présente, de manière informelle, la structure et le comportement de la classe BirdsActor, dont l'unique instance aBirdsActor est le système de capteurs de notre système de capture de mouvements, c'est-à-dire le dispositif contenant les capteurs et d'autres composants nécessaires pour pouvoir mesurer leur position et leur orientation. La section A1.5 présente également le type DATA qui permet à aBirdsActor de communiquer à aStreamActor les mesures concernant les capteurs.

Bien qu'aucun moyen de communication entre aBirdsActor et l'utilisateur ne soit représenté, c'est ce dernier qui «lance» et qui «détruit» aBirdsActor.

A1.5.1. Structure

La structure de la classe BirdsActor est présentée à la figure A1.13.

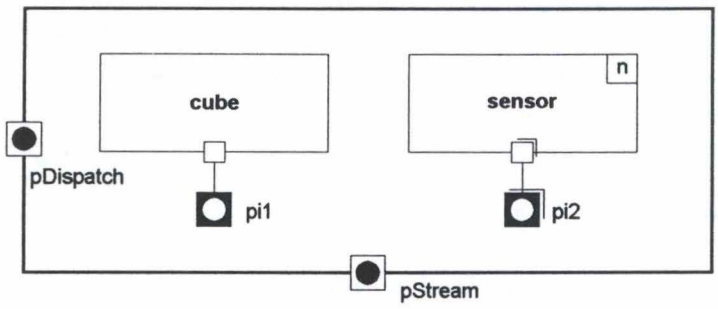


Figure A1.13 Structure de aBirdsActor

Toute instance de BirdsActor dispose de 2 portes, lui permettant de communiquer respectivement selon les protocoles PStreamBirds et PDispatchBirds. Dans notre modèle, la porte pStream de la seule instance aBirdsActor est reliée à aStreamActor, tandis que la porte pDispatch est reliée à aDispatchActor.

Les portes intérieures pi1 et pi2[x], $1 \leq x \leq n$, sont utilisées par aBirdsActor pour communiquer avec les acteurs qui sont contenus à l'intérieur de lui-même. Ceux-ci sont au nombre de n +1 — un «cube» et n capteurs. Le système disposait de 3 capteurs quand le code du système de capture de mouvements a été écrit. La structure et les fonctionnalités de ces acteurs contenus sont examinées ci-dessous (points A1.5.1.1 et A1.5.1.2).

Bien que cela ne soit pas représenté sur la figure A1.13, aBirdsActor contient aussi une variable sampleRate qui détermine le nombre de mesures qu'il effectue par seconde concernant ses capteurs.

A1.5.1.1. 'cube'

cube, de la classe CUBE, est un dispositif ayant la forme d'un cube et qui génère un champ électromagnétique dont le volume correspond approximativement à une sphère, dont le centre est le centre de cube et dont le diamètre est de 144 pouces, soit un peu plus de 3,65 mètres.

A1.5.1.2. 'sensor[x]'

Chaque instance **sensor[x]**, $1 \leq x \leq n$, de la classe **SENSOR**, représente un capteur du système, c'est-à-dire un dispositif qui, quand il se situe à l'intérieur de la sphère générée par **cube**, permet à **aBirdsActor** de mesurer sa position et son orientation. Ces mesures expriment en fait la position et l'orientation d'un *frame* local au capteur par rapport à un *frame* global, dont le centre est positionné au centre du cube. Ces *frames* sont représentés sur la figure A1.14.

Les deux exemples qui suivent montrent comment calculer la position et l'orientation d'un capteur :

- La position du capteur 1 des figures A1.15 et A1.16 est (d, 0, 0) : le centre du *frame* du capteur 1 est situé à une distance d sur l'axe x du *frame* du cube, et à une distance nulle sur les axes y et z de ce dernier.
- L'orientation du capteur 2 des figures A1.15 et A1.16 est (0, α, 0) : les plans xz du *frame* du capteur et du *frame* du cube sont parallèles, mais un angle α est formé entre leurs axes x et z.

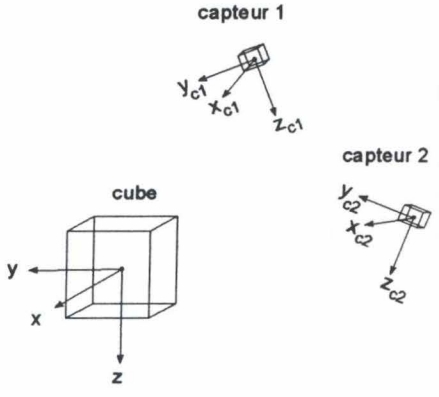


Figure A1.14
Frames du cube et des capteurs

6 valeurs sont nécessaires pour déterminer la position et l'orientation d'un capteur : on dit que celui-ci a 6 *degrés de liberté*.

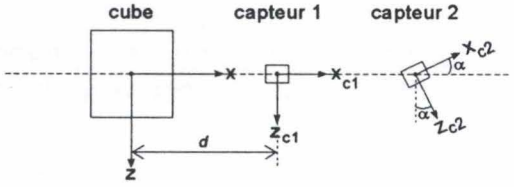


Figure A1.15
Vue de côté du cube et de 2 capteurs

Pour un **sensor[x]** donné, **aBirdsActor** peut décider de mesurer uniquement sa position, uniquement son orientation, les deux ensemble ou encore aucun des deux.

De plus, **aBirdsActor** est capable de calculer, sur base des 3 valeurs d'angles, une matrice 4x4 déterminant, comme les 3 valeurs, l'orientation du *frame* local du capteur par rapport au *frame* global du cube.

A chaque instance **sensor[x]** est donc associé un format de données, qui détermine le type de données qui sont mesurées par **aBirdsActor** pour cette instance. Les formats possibles sont ceux qui sont définis par les constantes:

- FLOCK_NOBIRDDATA (aucune donnée),
- FLOCK_POSITION (position uniquement),
- FLOCK_ANGLES (orientation uniquement, sous forme d'angles),
- FLOCK_MATRIX (orientation uniquement, sous forme de matrices),
- FLOCK_POSITIONANGLES (position *et* orientation sous forme d'angles) et
- FLOCK_POSITIONMATRIX (position *et* orientation sous forme de matrice).

Ces constantes ont déjà été définies au point A1.2.6.1.

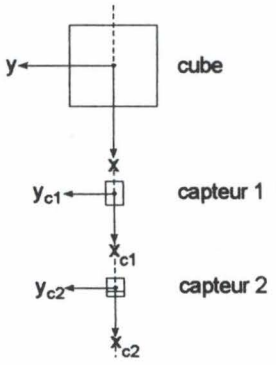


Figure A1.16
Vue de haut du cube et de 2 capteurs

A1.5.2. Comportement

Quand il reçoit de **aStreamActor** (sur sa porte **pStream**) un message de type **{start}** :

1• **aBirdsActor**, selon le format de données de chaque capteur, mesure la position et/ou l'orientation de chaque capteur (ou encore ne réalise aucune mesure) et stocke toutes ces mesures dans une variable de type DATA dont la structure fait l'objet du point A1.5.3 ;
2• **aBirdsActor** envoie cette variable à **aStreamActor** dans un message de type **{send_data, DATA}**,
3• il réitère les étapes 1 et 2 à un rythme de **sampleRate** fois par seconde, jusqu'à ce qu'il reçoive de **aStreamActor** un message **{stop}**.

Quand il reçoit de **aDispatchActor** (sur sa porte **pDispatch**) un message de type **{connect}**, **aBirdsActor** répond par un **{connect_confirm, int}**, la variable de type **int** étant égale à 0 si **aBirdsActor** et ses acteurs contenus sont tous en parfait état de fonctionnement. **aBirdsActor** ignore tous les messages arrivant sur ses portes **pDispatch** et **pStream** tant qu'un message de type **{connect}** ne lui est pas parvenu sur sa porte **pDispatch**.

Quand il reçoit de **aDispatchActor** un message de type **{ask_sensors_number}**, il répond par un **{sensors_number, int}**, la variable de type **int** contenant le nombre d'instances **sensor[x]** contenues dans **aBirdsActor**.

Quand il reçoit de **aDispatchActor** un message de type **{modif_config, CONFIG}**, il modifie sa variable **sampleRate** et les formats de données relatifs à chaque instance **sensor[x]** sur base de la variable de type **CONFIG** accompagnant le message.

Quand il reçoit de **aDispatchActor** un message de type **{disconnect}**, **aBirdsActor** «déconnecte» ses portes **pDispatch** et **pStream** : il ignore alors tous les messages provenant de ces portes jusqu'à ce qu'un message de type **{connect}** arrive sur sa porte **pDispatch**.

A1.5.3. Le type DATA

La définition C du type DATA, telle qu'elle avait été présentée dans le point A1.2.6.2 sans trop de commentaires, était la suivante:

```
typedef struct
{ HEADER          head;
  unsigned char  *buffer;
}DATA;
```

Pour pouvoir expliquer la structure de DATA, il nous faut aborder de manière un peu prématurée la manière dont la liaison entre **aStreamActor** et **aBirdsActor** est implémentée (figure A1.17).

Les implémentations de **aStreamActor** (un processus UNIX) et de **aBirdsActor** (la partie applicative du système MotionStar) communiquent via un protocole BIRDNET. En fait, tout processus s'adressant

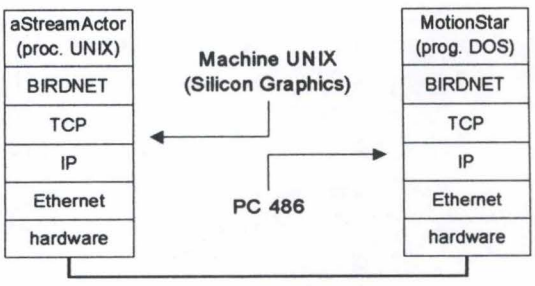


Figure A1.17
Implémentation de
aStreamActor et de aBirdsActor

au système MotionStar doit lui parler en BIRDNET. Ce protocole est basé sur TCP.

Le protocole PStreamBirds constitue une description ROOM de BIRDNET. PStreamBirds est beaucoup plus simple que BIRDNET : il n'envisage que les aspects de celui-ci qui ont été nécessaires pour l'écriture de notre système de capture de mouvements.

L'envoi d'un message de données PStreamBirds {send_data, DATA} correspond à l'envoi d'un message BIRDNET MSG_DATA_CONTINUED. Le paquet BIRDNET utilisé pour envoyer ce type de message correspond exactement au type DATA :

- head, de type HEADER, contient notamment le type du message (MSG_DATA_CONTINUED dans ce cas-ci), un code d'erreur, la version du protocole BIRDNET utilisée, un numéro de séquence, etc. Nous ne détaillerons pas la classe HEADER.

- buffer pointe vers un ensemble de bytes dont la longueur varie en fonction de la configuration de chaque capteur. La structure de buffer est illustrée à la figure A1.18.
  (1) désigne 1 byte qui contient le numéro du capteur;
  (2) désigne 1 byte qui contient le format de données du capteur ;
  (3) contient les données envoyées par le capteur. Par exemple :

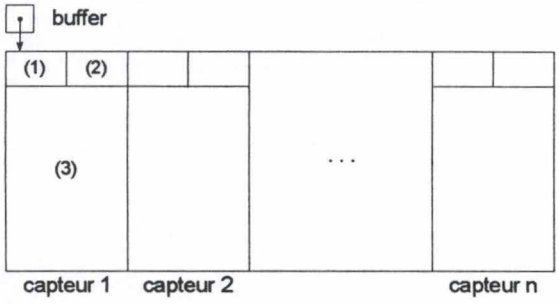


Figure A1.18 Le champ buffer de DATA

- si le format est FLOCK_POSITION (3 valeurs, en pouces, doivent être présentes), (3) contient 6 bytes (2 par valeur) ;

- si le format est FLOCK_ANGLES (3 valeurs d'angles, en degrés, doivent être présentes), (3) contient 6 bytes (2 par valeur) ;

- si le format est FLOCK_POSITIONMATRIX (3 valeurs, en pouces, + les 9 valeurs significatives de la matrice de rotation), (3) contient 24 bytes (2 par valeur).

2 bytes sont toujours utilisés pour coder une valeur, qu'elle soit en pouces, en degrés ou qu'elle provienne d'une matrice. La figure A1.19 montre comment interpréter ces 2 bytes selon le type de valeur qu'ils codent.

Valeurs min et max des 2 bytes (interprétés comme représentant une valeur entière)	-32 768	+ 32 767
Correspondance pour une valeur d'angle	-180°	+180°
Correspondance pour une valeur de distance	-144 pouces	+144 pouces
Correspondance pour une valeur de matrice	-1	+1

Figure A1.19
Interprétation des 2 bytes représentant une valeur provenant d'un capteur

A1.6. La classe d'acteur 'StreamActor'

Le but de la seule instance aStreamActor de la classe StreamActor est de formater les données de type DATA incluses dans les messages qui lui arrivent de aBirdsActor (messages de type {send_data, DATA}) et de transmettre ces données à aMotionActor (via des messages de type {send_data, char*} dans un format compréhensible par ce dernier. En effet, les valeurs de distances, d'angles et de matrices contenues dans les variables de type DATA sont des valeurs réelles qui sont chacune codées sur 2 bytes. Telles quelles, il est impossible de les manipuler : il faut les convertir dans un type sur lequel on peut faire des opérations, par exemple le type double du langage C, qui code un réel sur 8 bytes. C'est ce que fait aStreamActor.

La figure A1.20 présente la structure et le comportement de la classe d'acteur StreamActor.

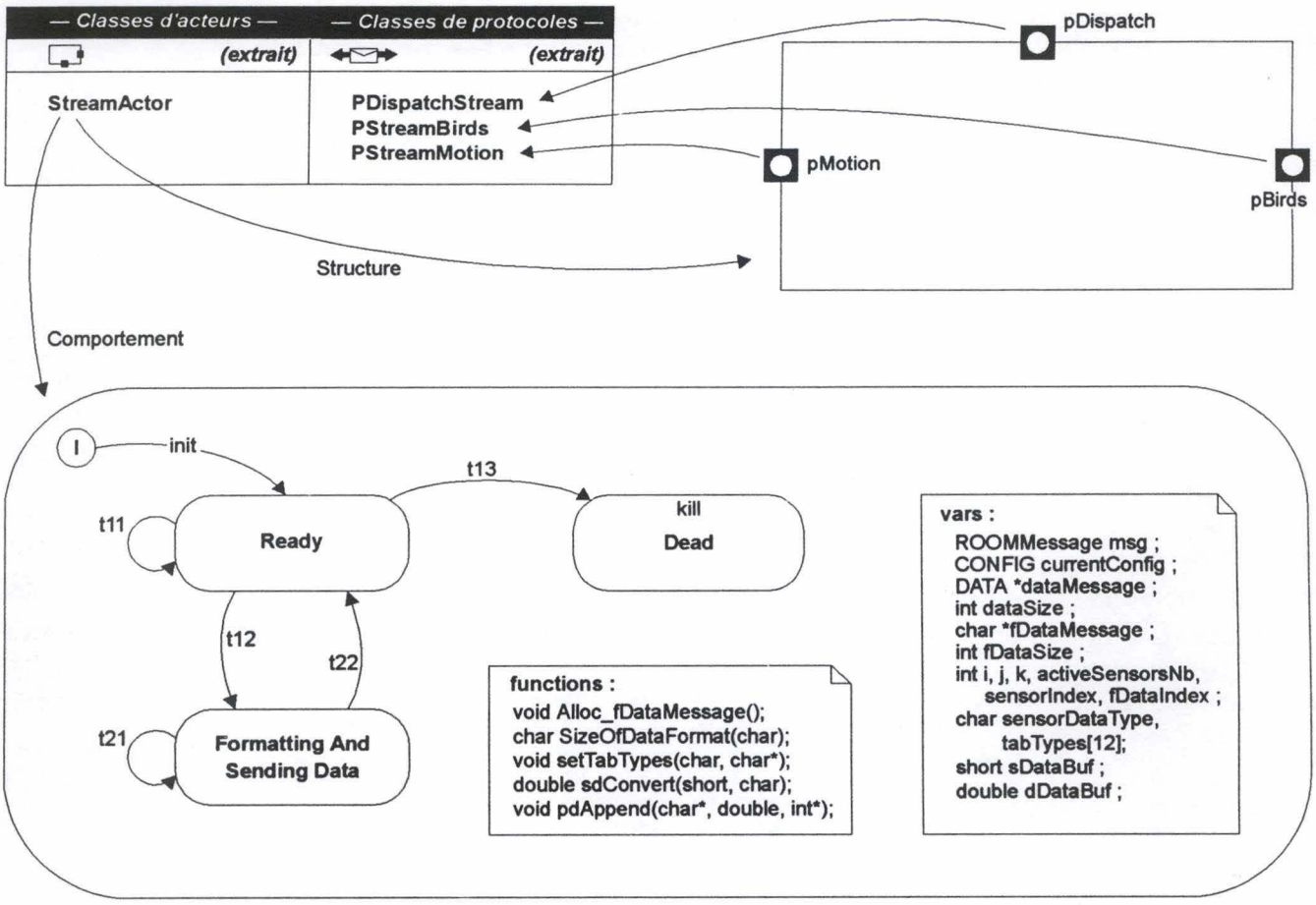


Figure A1.20 Structure et comportement de StreamActor

A1.6.1. Structure

Toute instance de StreamActor dispose de 3 portes lui permettant de communiquer selon les protocoles indiqués par les flèches. Dans notre modèle, les portes de la seule instance aStreamActor

de la classe StreamActor sont reliées à aBirdsActor (porte pBirds), aDispatchActor (porte pDispatch) et aMotionActor (porte pMotion).

A1.6.2. Comportement

Dans cette section, nous allons expliciter tous les éléments qui apparaissent sur le ROOMChart (le diagramme états-transitions) de la figure A1.20. On trouvera dans le point A1.6.2.1 un commentaire concernant les états du ROOMChart ; quant aux variables, fonctions et transitions, elles seront commentées et définies dans le point A1.6.2.2.

A1.6.2.1. Etats

aStreamActor peut se trouver dans trois états différents :

- l'état Ready (l'état initial), dans lequel aStreamActor se trouve juste après s'être initialisé;
- l'état Formatting and Sending Data, dans lequel aStreamActor reçoit des messages de aBirdsActor, les formate et les réexpédie à aMotionActor ;
- l'état Dead (l'état final), dans lequel aStreamActor termine son exécution.

A1.6.2.2. Transitions, variables et fonctions

transition init :

```
1    action :
2    { short sensorsNb ;
3      sensorsNb = msg->data[0] << 8;
4      sensorsNb += msg->data[1];
5      currentConfig.dataFormats = (char*) malloc(sensorsNb);
6      memcpy(currentConfig, msg->data, sensorsNb+1+sizeof(short));
7      Alloc_fDataMessage();
8    }
```

* aStreamActor est une instance optionnelle, incarnée par aDispatchActor. Lors de cette incarnation, aStreamActor reçoit un premier message, qui est copié dans msg et qui contient la configuration de aBirdsActor sous la forme d'une variable de type CONFIG (msg->data est un pointeur vers cette variable).

* Le contenu de msg->data est copié dans currentConfig, qui contient donc la configuration courante de aBirdsActor (ligne 6). Pour pouvoir allouer (ligne 5) à currentConfig l'espace dont il a besoin, aStreamActor détermine cet espace grâce aux deux premiers bytes de (msg->data)*, qu'il copie dans une variable temporaire sensorsNb (lignes 2 à 4).

* aStreamActor alloue ensuite l'espace nécessaire à la variable fDataMessage (ligne 7 : appel à la fonction Alloc_fDataMessage ci-dessous). fDataMessage contiendra les données des capteurs, provenant de aBirdsActor, après avoir été formatées : c'est donc cette variable qui sera envoyée à pMotionInterface dans un message de type {send_data, char*}.

void Alloc_fDataMessage()

```
1    { dataSize = 0 ;
2      activeSensorsNb = 0 ;
3      for (i=0; i<sensorsNb; i++)
4      {   dataSize += SizeOfDataFormat(currentConfig.dataFormats[i]);
5          if (currentConfig.dataFormats[i] != FLOCK_NOBIRDDATA) activeSensorsNb += 1;
6      }
7      fDataMessage = (char*) malloc (4*dataSize);
8    }
```

* Alloc_fDataMessage, lignes 1, 3 et 4, calcule dans dataSize la taille (en bytes) des données qui vont être envoyées par aBirdsActor à aStreamActor dans la variable de type DATA d'un message {send_data, DATA}. Pour ce faire, elle fait appel à la fonction SizeOfDataFormat (ci-dessous) qui, quand on lui donne un format de données relatif à un capteur, renvoie le nombre de bytes de données qui seront envoyés par ce capteur.
* Sur base de dataSize, Alloc_fDataMessage détermine, ligne 7, l'espace nécessaire à allouer à fDataMessage. Comme celui-ci, pour chaque valeur envoyée par un capteur, réserve 8 bytes (la taille d'une variable réelle de type double en C) et que aBirdsActor, dans son message de données, n'en réserve que 2, il faut multiplier dataSize par 4 pour obtenir l'espace à allouer à fDataFormat.
* Alloc_fDataMessage détermine aussi, lignes 2, 3 et 5, le nombre de capteurs qui sont «actifs» —ceux dont le format de données est différent de FLOCK_NOBIRDDATA.

char SizeOfDataFormat(char cFormat)

```
1    {     return 2*( (cFormat>>4) & 0x0F );
2    }
```

* Si, par exemple, cFormat = FLOCK_POSITIONMATRIX, SizeOfDataFormat renvoie 24 (le même exemple est expliqué plus en détail au point A1.5.3).

transition t11 :

```
1    triggered by : { {modif_config, pDispatch} or {stop, pDispatch} }
2    action :
3    { if (msg->signal == modif_config)
4      {   memcpy(currentConfig, msg->data, currentConfig.sensorsNb + 1 + sizeof(short);
5          free(fDataMessage);
6          Alloc_fDataMessage();
7      }
8    }
```

* Quand aStreamActor reçoit une nouvelle configuration, il la copie dans sa variable currentConfig (ligne 4). Puisque la configuration a changé, il est très probable que la taille des données en provenance de aBirdsActor n'est plus la même, et donc que la taille de fDataMessage n'est plus adaptée. aStreamActor désalloue alors fDataMessage et le réalloue en fonction de la nouvelle configuration (lignes 5 et 6).

transition t12 :

```
1    triggered by : { {start, pDispatch} }
2    action :
3    { pBirds.send(start) ;
4    }
```

* Quand aStreamActor reçoit un message {start} de aDispatchActor, il le transmet à aBirdsActor pour que celui-ci lui envoie des messages de données.

transition t13 :

```
1    triggered by : { {kill, pDispatch} }
2    action : -
```

* Quand aDispatchActor signifie à aStreamActor que le système doit être clôturé, aStreamActor passe dans l'état Dead. Aucune action n'est associée à t13.

transition t21 :

```
1    triggered by : { {send_data, pBirds} }
2    action :
3    { dataMessage = (DATA *) msg->data ;
4      i = 0 ; sensorIndex = 1 ; fDataIndex = 0;
5      while (i++ < activeSensorsNb)
6      {   sensorDataType = (dataMessage->buffer[sensorIndex] >> 4 & 0x0F;
```

```
7           sensorDataType += (dataMessage->buffer[sensorIndex] << 4) & 0xF0;
8           setTabTypes(sensorDataType, tabTypes);
9           k = 0;
10          for (j=sensorIndex+1 ; j < (sensorIndex+1+SizeOfDataFormat(sensorDataType)); j+= 2)
11          {       sDataBuf = dataMessage->buffer[j];
12                  sDataBuf <<= 8;
13                  sDataBuf += dataMessage->buffer[j+1];
14                  dDataBuf= sdConvert(sDataBuf, tabTypes[k++]);
15                  pdAppend(fDataMessage, dDataBuf, &fDataIndex);
16          }
17          sensorIndex = j + 1;
18      }
19      pMotion.send(send_data, fDataMessage);
20  }
```

* L'objectif de aStreamActor, quand il reçoit un message {send_data, DATA} de aBirdsActor, est de remplir fDataMessage, sur base de dataMessage –qui, après la ligne 3, pointe vers la structure DATA contenant les données « brutes » provenant de aBirdsActor–, et de l'envoyer à aMotionActor via un message de type {send_data, char*}. La figure A1.21 illustre la structure de dataMessage et de fDataMessage.

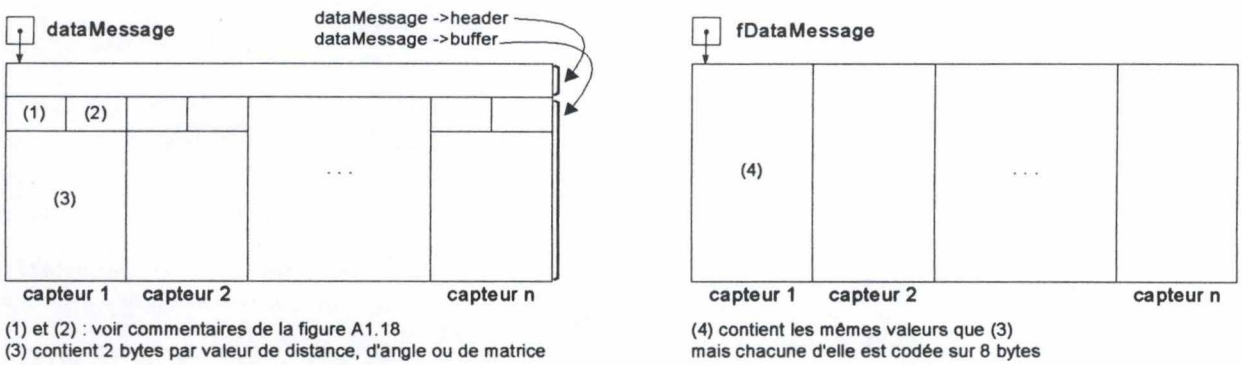


Figure A1.21 De dataMessage à fDataMessage

* A chaque itération de la boucle while (lignes 5 à 18), les données concernant un capteur sont ajoutées à fDataMessage. Quand aStreamActor sort de la boucle, fDataMessage est rempli: il peut être envoyé à aMotionActor (ligne 19). Voyons en détail une itération de cette boucle:

- lignes 6 à 8 : aStreamActor copie d'abord dans sensorDataType le format de données du capteur courant (il le copie à partir de dataMessage->buffer mais aurait tout aussi bien pu le faire à partir de currentConfig.dataFormats). Il utilise sensorDataType pour initialiser un «tableau de types» tabTypes (appel à la fonction setTabTypes ci-dessous) qui contient le type de chaque valeur (distance, angle ou matrice) envoyée par le capteur en fonction de son format de données. La figure A1.22 illustre le tableau de types pour un capteur dont le format de données est FLOCK_POSITIONMATRIX.
- lignes 9 à 17 : pour le capteur courant, à chaque itération de la boucle for, une valeur de distance, d'angle ou de matrice provenant de dataMessage est convertie et ajoutée à fDataMessage. Voyons en détail une itération de cette boucle:
  - lignes 11 à 13 : la valeur courante du capteur courant est copiée de dataMessage dans un buffer sDataBuf de 2 bytes.
  - ligne 14 : cette valeur est convertie (appel à la fonction sdConvert ci-dessous) en valeur réelle de 8 bytes et stockée dans un buffer dDataBuf, de 8 bytes. Pour faire la

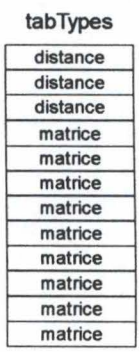


Figure A1.22
tabTypes pour le format FLOCK_POSITIONMATRIX

conversion, il faut bien sûr savoir de quel type est la valeur (distance, angle ou matrice) : ce type est fourni par l'élément du tableau de types tabTypes qui a trait à la valeur courante.
• ligne 15 : la valeur réelle dDataBuf est ajoutée à fDataMessage (appel à la fonction pdAppend ci-dessous).

```
void setTabTypes(char cType, char *tabTypes)
1    { int i;
2      switch (cType)
3      {   case FLOCK_POSITION :
4                for (iLoop=0; iLoop<3; iLoop++) tabTypes[iLoop]=POSITION_DATA;
5                break;
6          case FLOCK_ANGLES :
7                for (iLoop=0; iLoop<3; iLoop++) tabTypes[iLoop]=ANGLES_DATA;
8                break;
9          case FLOCK_MATRIX :
10               for (iLoop=0; iLoop<9; iLoop++) tabTypes[iLoop]=MATRIX_DATA;
11               break;
12         case FLOCK_POSITIONANGLES :
13               for (iLoop=0; iLoop<3; iLoop++) tabTypes[iLoop]=POSITION_DATA;
14               for (iLoop=3; iLoop<6; iLoop++) tabTypes[iLoop]=ANGLES_DATA;
15               break;
16         case FLOCK_POSITIONMATRIX :
17               for (iLoop=0; iLoop<3; iLoop++) tabTypes[iLoop]=POSITION_DATA;
18               for (iLoop=3; iLoop<12; iLoop++) tabTypes[iLoop]=MATRIX_DATA;
19               break;
20     }
21   }
```

* setTabTypes, en fonction du format cType qui lui est donné en argument, remplit tabTypes. Celui-ci doit avoir été alloué avant d'être donné comme argument à setTabTypes. Dans notre cas, aStreamActor fournit toujours à setTabTypes un tableau de 12 éléments.
* Lignes 3 à 5 : cType = FLOCK_POSITION : seuls les 3 premiers éléments de tabTypes sont remplis, avec la valeur POSITION_DATA (une valeur de distance en pouces).
* Lignes 16 à 19 : cType = FLOCK_POSITIONMATRIX : les 12 éléments de tabTypes sont remplis: les 3 premiers le sont avec la valeur POSITION_DATA, les 9 derniers avec la valeur MATRIX_DATA.

```
double sdConvert(short sDataBuf, char cType)
1    { double dReturnValue;
2      switch (cType)
3      {   case POSITION_DATA :
4                dReturnValue = (double) (sDataBuf/227.56441);
5                break;
6          case ANGLES_DATA :
7                dReturnValue = (double) (sDataBuf/182.049);
8                break ;
9          case MATRIX_DATA :
10               dReturnValue = (double) (sDataBuf/32768.311);
11               break ;
12     }
13     return dReturnValue;
14   }
```

* sdConvert, selon le type cType, convertit la valeur sDataBuf (2 bytes) en double (8 bytes). Les valeurs par lequelles il faut diviser sDataBuf (lignes 4, 7 et 10) pour obtenir la valeur correcte se déduisent de la figure A1.19.

```
void pdAppend(char *ptPacketBuf, double dDataBuf, int *iIndice)
1    { memcpy(&ptPacketBuf[*iIndice], &dDataBuf, sizeof(double) );
2      *iIndice += sizeof(double);
3    }
```

* pdAppend ajoute à la fin de ptPacketBuf la valeur réelle dDataBuf. L'endroit auquel il faut ajouter dDataBuf est déterminé par iIndice. Celui-ci est mis à jour après l'ajout (pour l'ajout suivant) et renvoyé comme paramètre résultat.

transition t22 :

```
1    triggered by : { {stop, pDispatch} }
2    action :
3    { pBirds.send(stop) ;
4    }
```

* Quand aStreamActor reçoit un message {stop} de aDispatchActor, il le transmet à aBirdsActor pour que celui-ci arrête de lui envoyer des messages de données.

entry action kill :

```
{    free(fDataMessage);
}
```

* Avant de terminer son exécution, aStreamActor désalloue fDataMessage.

A1.7. La classe d'acteur 'DispatchActor'

Le but de aDispatchActor, seule instance de la classe DispatchActor, est double :

- son premier but est de relayer les messages qui proviennent de aMotionActor à destination de aBirdsActor et aStreamActor ;

- son second est d'assurer, pour une large part, l'initialisation et la destruction du système de capture de mouvements. Bien qu'aucun moyen de communication ne soit représenté entre aDispatchActor et l'utilisateur, c'est ce dernier qui «lance» aDispatchActor. Celui-ci se connecte ensuite à aBirdsActor (si celui-ci a été au préalable «lancé» par l'utilisateur), puis, dans le cas d'une connexion réussie, «crée» aStreamActor et aMotionActor, ce dernier créant à son tour anInterfaceActor. L'ordre enjoignant de «détruire» le système, on le sait, provient de l'utilisateur, via anInterfaceActor, cette fois-ci. Quand aDispatchActor reçoit cet ordre de aMotionActor, il le transmet à aStreamActor et se déconnecte de aBirdsActor, qui doit être «détruit» séparément par l'utilisateur.

La figure A1.23 présente la structure et le comportement de la classe d'acteur DispatchActor.

A1.7.1. Structure

Toute instance de DispatchActor dispose de 3 portes lui permettant de communiquer selon les protocoles indiqués par les flèches. Dans notre modèle, les portes de la seule instance aDispatchActor de la classe DispatchActor sont reliées à aBirdsActor (porte pBirds), aStreamActor (porte pStream) et aMotionActor (porte pMotion).

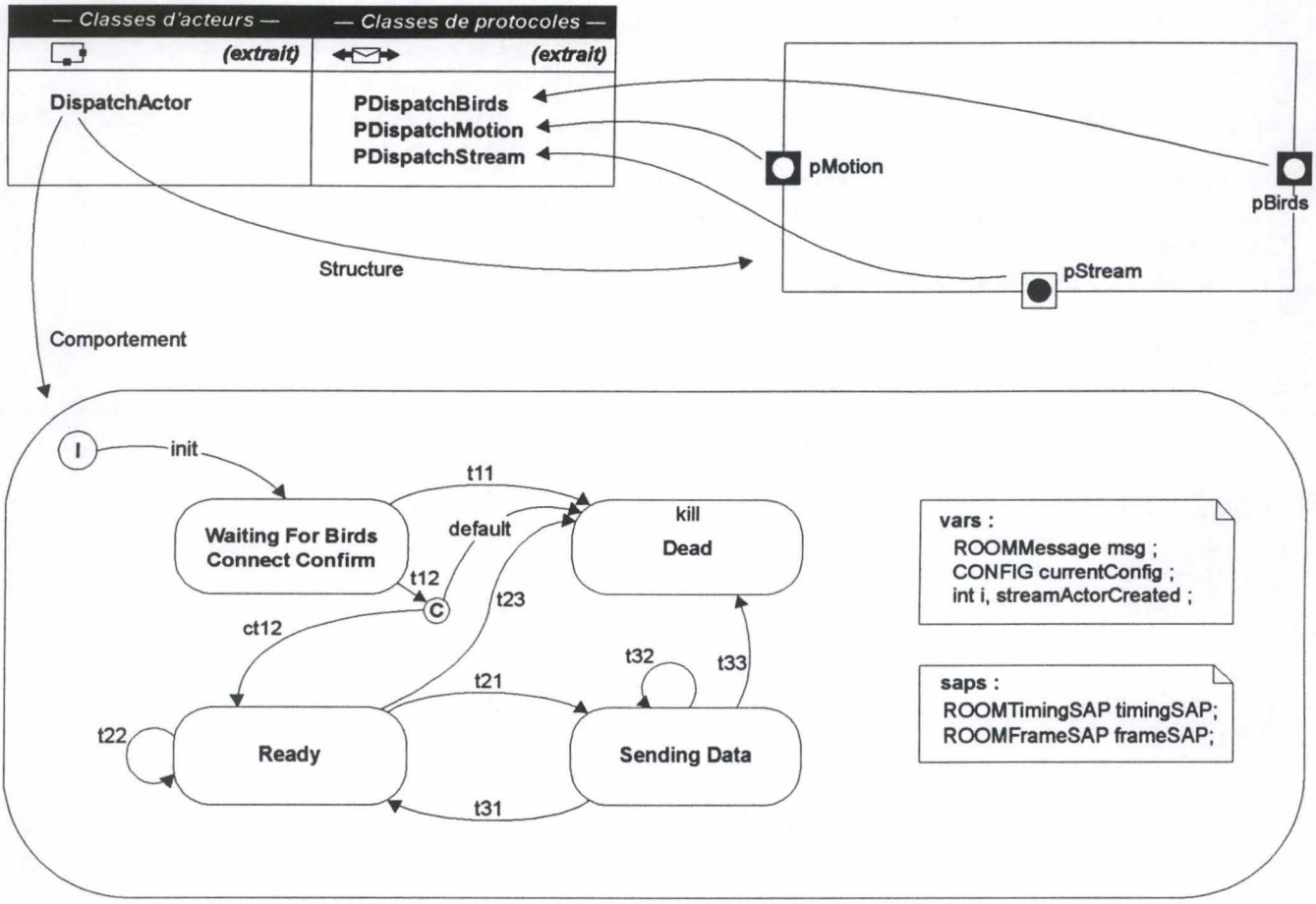


Figure A1.23 Structure et comportement de DispatchActor

A1.7.2.Comportement

Dans cette section, nous allons expliciter tous les éléments qui apparaissent sur le ROOMChart (le diagramme états-transitions) de la figure A1.23. On trouvera dans le point A1.7.2.1 un commentaire concernant les états du ROOMChart. Les variables et SAPs seront commentés dans le point A1.7.2.2; nous terminerons par la définition des transitions dans le point A1.7.2.3.

A1.7.2.1. Etats

aDispatchActor peut se trouver dans 4 états différents :

- l'état Waiting for Birds Connect Confirm (l'état initial), dans lequel il attend que aBirdsActor réponde à sa demande de connexion ;
- l'état Ready, dans lequel il est prêt à recevoir des messages ;
- l'état Sending Data, dans lequel aStreamActor est en train de recevoir des données de aBirdsActor. aDispatchActor dispose de cet état car il doit tenir compte de l'état de aStreamActor quand on lui demande de traiter une modification de configuration ;
- l'état Dead (l'état final), dans lequel aDispatchActor termine son exécution.

A1.7.2.2. Variables et SAPs

aDispatchActor dispose des variables et SAPs suivants :

- timingSAP est un SAP qui lui permet de communiquer avec le *timing service* de ROOM. aDispatchActor peut par exemple demander au *timing service* de lancer un *timer* et de le prévenir quand celui-ci arrive à son terme.
- frameSAP est un SAP qui permet à aDispatchActor de communiquer avec le *frame service* ROOM. aDispatchActor peur demander au *frame service* de créer un autre acteur en lui donnant un message d'initialisation.
- msg est une variable ROOM de type ROOMMessage qui contient toujours le dernier message que l'acteur a reçu.
- currentConfig, de type CONFIG, contient la configuration courante de aBirdsActor.
- streamActorCreated est un booléen indiquant si aStreamActor est déjà créé ou non.
- i est un compteur.

A1.7.2.3. Transitions

transition init :

```
1    action :
2    { pBirds.send(connect);
3      timingSAP.informIn(2000);
4       streamActorCreated = 0;
5    }
```

* Quand l'utilisateur «lance» aDispatchActor, celui-ci franchit sa transition initiale init: il tente de se connecter à aStreamActor (ligne 2), demande au *timing service* de le prévenir dans 2000 millisecondes (ligne 3) et met streamActorCreated à FALSE.

transition t11 :

```
1    triggered by : { {timeout, timingSAP} }
2    action : -
```

* Si aDispatchActor reçoit un message {timeout} sur son timingSAP, c'est qu'il a attendu 2 secondes depuis l'envoi du message {connect} (transition init) sans recevoir de réponse. Il n'attend pas plus et passe dans l'état Dead.

transition t12 :

```
1    triggered by : { {connect_confirm, pBirds} }
2    branch condition ct12 : (msg->data == 0)
3          action :
4          { timingSAP.send(cancelTimer);
5            msg = pBirds.invoke(ask_sensors_number);
6            currentConfig.sensorsNb = ((int *) msg->data)* ;
7            currentConfig.sampleRate = 25 ;
8            currentConfig.dataFormats = (char *) malloc (currentConfig.sensorsNb);
9           currentConfig.dataFormats[0] = FLOCK_POSITIONMATRIX ;
10           for (i=1; i<currentConfig.sensorsNb; i++)
11               currentConfig.dataFormats[i] = FLOCK_MATRIX;
12          pBirds.send(modif_config, currentConfig);
13          frameSAP.incarnate(aStreamActor, currentConfig);
14           streamActorCreated = 1;
15          frameSAP.incarnate(aMotionActor, currentConfig);
16         }
```

```
17    default :
18          action :
19          { timingSAP.send(cancelTimer);
20          }
```

* Si aDispatchActor reçoit un message de type {connect_confirm, int}, c'est que aBirdsActor a répondu à son message {connect} (transition init) avant les 2 secondes. Le *timer* n'est donc plus nécessaire et, quel que soit le résultat de la condition de branchement du *choicepoint*, aDispatchActor annule le *timer* (lignes 4 et 19).
* Si la variable de type int accompagnant le message est nulle (ct12 est TRUE) :
  - aDispatchActor demande à aBirdsActor le nombre de capteurs (ligne 5), qui lui est nécessaire pour remplir currentConfig (lignes 6 à 11) avec la configuration par défaut. Cette configuration est :
    - un *sample rate* de 25 ;
    - le format de données FLOCK_POSITIONMATRIX pour le premier capteur ;
    - le format de données FLOCK_MATRIX pour les autres.
  - aDispatchActor envoie cette configuration à aBirdsActor (ligne 12) ;
  - aDispatchActor crée aStreamActor et aMotionActor, en leur donnant comme donnée d'initialisation la configuration courante de aBirdsActor (lignes 13 à 15).
  - aDispatchActor passe dans l'état Ready.
* Si la variable de type int accompagnant le message est non nulle (ct12 est FALSE), aDispatchActor passe dans l'état Dead.

transition t21 :

```
1     triggered by : { {start, pMotion} }
2     action :
3     { pStream.send(start);
4     }
```

* Quand aDispatchActor reçoit de aMotionActor l'ordre de démarrer l'animation, il transmet le message à aStreamActor.

transition t22 :

```
1     triggered by : { {modif_config, pMotion} or {stop, pMotion} }
2     action :
3     { if (msg->signal == modif_config)
4       {   pBirds.send(modif_config, msg->data);
5           pStream.send(modif_config, msg->data);
6       }
7     }
```

* Quand aDispatchActor reçoit de aMotionActor l'ordre de remplacer la configuration courante par celle qu'il lui donne, alors qu'il est dans l'état Ready (le cas qui nous occupe ici), il transmet celle-ci à aBirdsActor et à aStreamActor.

transition t23 :

```
1     triggered by : { {kill, pMotion} }
2     action : -
```

* Quand aDispatchActor reçoit de aMotionActor l'ordre de clôturer le système, alors qu'il est dans l'état Ready, il passe dans l'état Dead sans effectuer d'action.

transition t31 :

```
1     triggered by : { {stop, pMotion} }
2     action :
3     { pStream.send(stop);
4     }
```

* Quand aDispatchActor reçoit de aMotionActor l'ordre de stopper l'animation, il transmet le message à aStreamActor.

transition t32 :

```
1    triggered by : { {modif_config, pMotion} or {start, pMotion} }
2    action :
3    { if (msg->signal == modif_config)
4      {   pStream.send(stop);
5          pBirds.send(modif_config, msg->data);
6          pStream.send(modif_config, msg->data);
7          pStream.send(start);
8      }
9    }
```

* Quand aDispatchActor reçoit de aMotionActor l'ordre de remplacer la configuration courante par celle qu'il lui donne, alors qu'il est dans l'état Sending Data :
  - il stoppe d'abord l'animation (ligne 4),
  - il transmet la nouvelle configuration à aBirdsActor et à aStreamActor (lignes 5 et 6),
  - il relance l'animation (ligne 7).

transition t33 :

```
1    triggered by : { {kill, pMotion} }
2    action :
3    { pStreamActor.send(stop);
4    }
```

* Quand aDispatchActor reçoit de aMotionActor l'ordre de clôturer le système, alors qu'il est dans l'état Sending Data, il stoppe d'abord l'animation et passe ensuite dans l'état Dead.

entry action kill :

```
1    { if streamActorCreated
2      {   pStream.send(kill);
3          pBirds.send(disconnect);
4          free(currentConfig.dataFormats);
5      }
6    }
```

* Avant de terminer son exécution, aDispatchActor, pour autant qu'il ait créé aStreamActor et qu'il ait alloué de l'espace pour currentConfig.dataFormats –ce qui n'est pas le cas si aBirdsActor n'a pas répondu à son message {connect} (transition init)–, il envoie un message {kill} à aStreamActor (ligne 2), se déconnecte de aBirdsActor (ligne 3) et désalloue l'espace qu'il avait alloué à currentConfig.dataFormats (ligne 4).